UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Analizador estático de código Ruby

Esteban García Bravo
Tutor: Jesús Sánchez Cuadrado
Ponente: Esther Guerra Sánchez

Julio 2015

RESUMEN

Ruby es un lenguaje de programación relativamente nuevo que día a día se está haciendo más popular. La principal diferencia que tiene con respecto a los lenguajes de programación tradicionales es su **carácter dinámico**. Esto supone ciertas **ventajas** de cara al usuario como la libertad de inyectar código en tiempo de ejecución o no tener que asignar un tipo estático a las variables de programa.

Esta libertad tiene un precio, y es que precisamente ese **tipado de variables** se hace en el propio **tiempo de ejecución** y por lo tanto hasta entonces no se puede saber si se ha cometido un error de ese tipo, mientras que en otros lenguajes tradicionales estos errores se detectan en **tiempo de compilación**.

Así pues, el objetivo de este trabajo es el desarrollo de un analizador de código Ruby capaz de **asignar tipos a las variables estáticamente** para así reportar los posibles errores cometidos **antes del tiempo de ejecución**.

El analizador es accesible a través de un **API**, con operaciones para ejecutar el análisis, reportar errores relativos al tipado y permitir consultas para proveer información acerca de los objetos de programa (clases, funciones, etc.) al usuario de forma clara y concisa.

En este documento veremos cómo se ha llevado a cabo el estudio, análisis, diseño, implementación y validación de este **analizador estático de código Ruby**, que abre la posibilidad para su posterior integración en un entorno de desarrollo real.

PALABRAS CLAVE

API, analizador, lenguaje dinámico, lenguaje estático, Ruby, tiempo de ejecución, tiempo de compilación, tipado dinámico, tipado estático.

ABSTRACT

Ruby is a relatively new programming language which is becoming more popular each day. The main difference between Ruby and other classic programming languages is its **dynamic behavior**. From the developer's point of view this implies several benefits such as the possibility of adding new code at runtime or not requiring the assignment of types to program variables statically.

This freedom comes at a price, which is that the **type checking** is made on **runtime** so the user cannot know until then if he has made mistakes about typing, while in other classic programming languages these mistakes are reported at **compilation time**.

Because of that, the main goal of this project is to develop a Ruby code analyzer capable of **typing objects statically** so it can detect and report errors **before runtime**.

This analyzer can be used through an **API** with options such as executing the analysis, reporting errors and providing all the information about program objects (classes, functions, etc.) to the user in a simple and clear way.

In this document we will explain how the study, analysis, design, implementation and validation of this **Ruby code static analyzer** has been carried out, allowing its integration with real development environments in the future.

KEYWORDS

API, analyzer, dynamic language, static language, program objects, Ruby, runtime, compilation time, dynamic typing, static typing.

Agradecimientos:

A mi tutor Jesús

A mis compañeros de curso

Y a mis amigos y mi familia

Gracias por el apoyo

ÍNDICE DE CONTENIDO

ÍNDICE DE FIGURAS

1. INTRODUCCIÓN

1.1 CONTEXTO Y MOTIVACIÓN

Ruby [1] es un lenguaje de programación con licencia de software libre relativamente joven, creado en 1995 por el desarrollador japonés **Yukihiro Matsumoto** y que no se dio a conocer en occidente hasta cierto tiempo después, ya que toda la documentación asociada estaba escrita precisamente en japonés. En el proceso de creación, se tomaron ciertas características de otros lenguajes ya existentes como **Python** o **Perl** como base, consiguiendo así un lenguaje bastante potente.

Esta potencia reside en la **versatilidad**, la libertad y en cierta medida en lo simple que resulta programar en él. En palabras de su propio creador, su intención fue "*crear un lenguaje de programación centrado en facilitar las cosas al usuario y no a la máquina*". Así pues, Ruby permite cierta **flexibilidad** a la hora de programar que otros lenguajes clásicos no tienen.

FIGURA 1: LOGOTIPO LENGUAJE RUBY

Si hablamos de características más concretas, podríamos decir que Ruby es un **lenguaje orientado a objetos**. Más que orientado a objetos, en Ruby absolutamente todo es un objeto, hasta los tipos de datos más básicos como los enteros o los booleanos. Así pues, siguiendo esta línea nos encontramos con que las variables son simplemente instancias de estos objetos ya mencionados, y que las funciones son métodos de estas clases, aunque ya hablaremos de ello en profundidad más adelante.

Ruby es además un **lenguaje interpretado** [2], una característica poco común en otros lenguajes de programación más populares. Esto significa que Ruby no tiene un compilador que traduzca las sentencias a lenguaje máquina antes de su ejecución, sino que el propio "intérprete" de Ruby es el encargado de traducir y ejecutar el programa al mismo tiempo.

Una desventaja que puede ocasionar este tipo de lenguaje es que la ejecución es mucho más lenta con respecto a los lenguajes compilados, sin embargo ya vimos qué opinaba el propio creador de Ruby sobre esto. En cambio las ventajas son bastante notables, ya que permite a Ruby acceder al grupo de los **lenguajes dinámicos** [3].

Los lenguajes dinámicos son aquellos que permiten ser modificados en el mismo tiempo de ejecución, ya sea añadiendo código nuevo, extendiendo algunas clases o editando el tipo de ciertas variables. Esto es debido a que a las variables se les asigna su tipo en tiempo de ejecución.

Respecto a esto, es interesante hablar sobre ***Duck Typing*** [4] ya que es el sistema de tipado que sigue Ruby. Simplificado en una frase, sería algo así como:

"Si un pájaro anda como un pato, nada como un pato y hace el mismo sonido que un pato, asumo que ese pájaro es un pato"

Si trasladamos esa idea al mundo de la programación, se puede entender que el tipado dinámico se basa en conocer los métodos de determinado objeto y asignar un tipo en base a que sea capaz de ejecutar esos métodos.

```
1  objeto Perro {
2      funcion ladrar() {
3          "Guau"
4      }
5  }
6
7  objeto Loro {
8      funcion ladrar() {
9        "Imita un Guau"
10     }
11 }
12
13 funcion sonido(perro) {
14     perro.ladrar()
15 }
16
```

FIGURA 2: PSEUDOCÓDIGO EXPLICATIVO SOBRE DUCK TYPING

En la **Figura 2** podemos ver un ejemplo sencillo para ilustrar cómo funciona el *Duck Typing*. Primero tenemos un objeto ***Perro*** el cual puede *ladrar* que se trata de emitir un "Guau". Después tenemos un objeto ***Loro*** que aunque no sea capaz de ladrar como tal, si que puede imitar el sonido del perro y emitir un "Guau" también.

Después tenemos una función ***sonido*** que hace ladrar a todo perro que reciba por parámetro. Si llamamos a esta función *sonido* pasándole el objeto *Perro* se oirá un "Guau" mientras que si le pasamos el objeto Loro obtendremos el mismo sonido. Por lo tanto, siguiendo la filosofía de *Duck Typing*, **a ojos del programa *Perro* y Loro son un objeto del mismo tipo**, cuando en realidad no lo son. Hablaremos más detenidamente de ésto y de los problemas que origina más adelante.

Este dinamismo provee una **potencia única** y una **libertad** a los programadores que resultan bastante atractivas y hace que muchos prefieran este tipo de lenguajes a los clásicos con tipado estático.

Sin embargo, no todo son ventajas ya que también presenta ciertos **puntos negativos**, siendo el principal el problema de **reconocer el tipo de las variables y objetos** que se están usando al programar, un problema que lleva persiguiendo a los lenguajes dinámicos desde su propia creación.

Ya hemos comentado que en los lenguajes dinámicos no hay declaración estática de los tipos de las variables y tampoco una compilación previa a la ejecución, por lo tanto todos los posibles errores de esta clase se detectan en el propio tiempo de ejecución.

Por lo tanto, la motivación de este proyecto es la creación de un **analizador capaz de detectar errores relacionados principalmente con el tipado** de todas las variables presentes en el código para que puedan ser corregidos antes de la ejecución.

1.2 EJEMPLOS EN RUBY

A continuación vamos a ejemplificar ciertas partes básicas de Ruby para entender cómo es la sintaxis del lenguaje de cara comprender mejor la posterior explicación acerca del funcionamiento del analizador.

Vamos a ver un ejemplo sencillo de **asignación** de valores a variables:

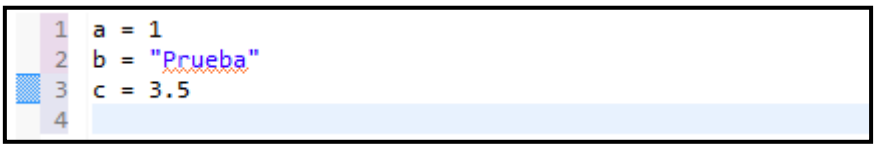


FIGURA 3: EJEMPLO DE ASIGNACIONES EN RUBY

En la **Figura 3** vemos como a la variable *a* se le asigna un valor de tipo *int*, a la variable *b* uno de tipo *string* y a la variable *c* uno de tipo *float*. En ninguno de los tres casos hemos necesitado inicializar las variables ni aclarar su tipo, como tendríamos que hacer en cualquier lenguaje con tipado estático.

Siguiendo la explicación del punto anterior sobre *Duck Typing*, Ruby al ver que por ejemplo a la variable *a* se le asigna un 1, asume que la variable es de tipo entero y lo mismo con el resto de variables.

Ahora vamos a ver un ejemplo de una **clase** sencilla en Ruby

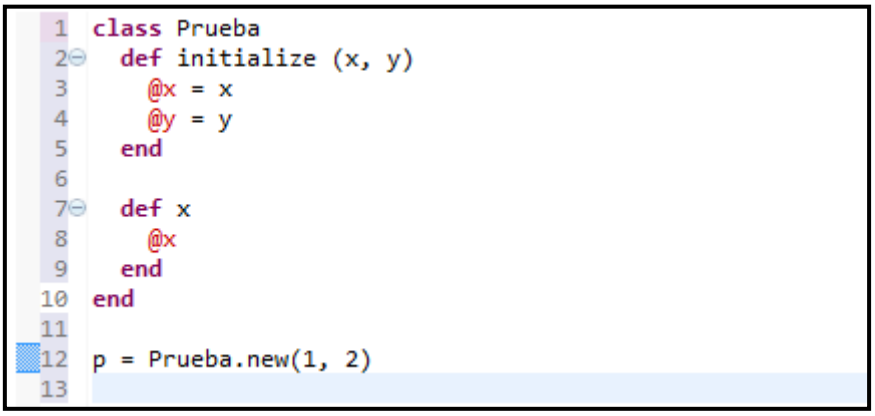


FIGURA 4: EJEMPLO DE CLASES EN RUBY

En la **Figura 4** *Prueba* es un objeto de tipo clase, *initialize* es su método constructor y *x* es otro método dentro de la clase. *@x* y *@y* son dos variables de instancia, de ahí que se le añada @ delante del nombre. Para acabar, tenemos *p* que es un objeto tipo *Prueba*.

Aprovechando este ejemplo para explicar cómo funcionan los métodos en Ruby, podemos ver que todas deben empezar por *def* y, como toda estructura en este lenguaje, acabar con *end*.

En este ejemplo tampoco es necesario aclarar el tipo de los argumentos de la función, y en este mismo caso si la sentencia final fuera *"p = Prueba.new("a", "b")"* no ocurriría ningún problema, lo único que cambiaría es que Ruby daría un tipo distintos a las variables de instancia de *p*.

Si por el contrario tuviéramos algo así:

```ruby
class Prueba
  def initialize (x, y)
    @x = x + 1
    @y = y
  end

  def x
    @x
  end
end

p = Prueba.new("a", "b")
```

FIGURA 5: OTRO EJEMPLO DE CLASES EN RUBY

Ruby tendría problemas con el ejemplo de la **Figura 5** ya que no sería capaz de asignarle un tipo correcto a *@x*. En este caso estaríamos mezclando un tipo *string* con un *int*, algo que no es posible en términos de programación. La diferencia está en que un lenguaje estático sería capaz de detectar este error antes de la ejecución ya que toda variable tendría su tipo definido, mientras que en Ruby se detecta en plena ejecución.

```ruby
numero = 8

if numero < 0
  es = "negativo"
elsif numero == 0
  es = "cero"
else
  es = "positivo"
end
```

FIGURA 6: EJEMPLO DE CONDICIONALES EN RUBY

En la **Figura 6** podemos ver un ejemplo de **condicional** en Ruby. La sintaxis es muy similar a otros lenguajes como C o Java sin embargo Ruby permite más libertad ya no solo en cuanto a declaración de variables (o en este caso la falta de declaración de la variable *es*), sino también en la opcionalidad de incluir paréntesis en las sentencias condicionales.

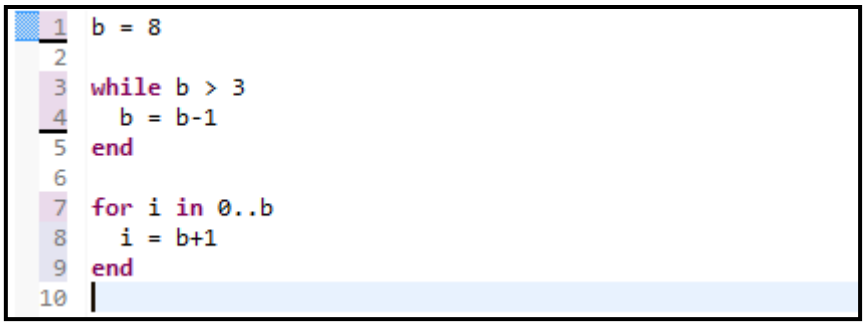


FIGURA 7: EJEMPLO DE BUCLES EN RUBY

Si vemos cómo es la sintaxis de los **bucles** en Ruby en la **Figura 7**, nos damos cuenta de que es prácticamente igual a la del resto de lenguajes de programación más populares. Lo único a destacar es en cuanto a los rangos del bucle *for*, en el que también se deben hacer comprobaciones para ver si las variables usadas tienen los tipos adecuados.

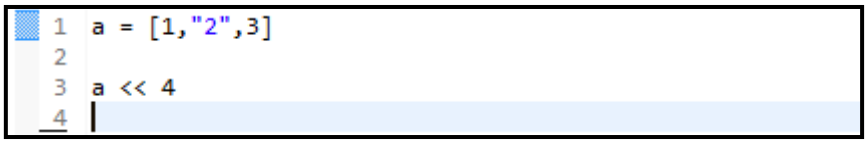


FIGURA 8: EJEMPLO DE ARRAYS EN RUBY

Si hablamos de ***arrays***, tenemos que hablar de nuevo de la libertad característica de Ruby. Si observamos la **Figura 8**, el *array a* tiene tres elementos, dos de ellos de tipo *int* y uno de tipo *string*. En los lenguajes tradicionales esto sería mucho más complicado de hacer ya que de antemano tenemos que asignarle un tipo al *array* (antes incluso de empezar a incluir elementos). Los *arrays* en Ruby no tienen un tipo definido, sino que se podría entender que su tipo es la **unión** de los tipos que lo conforman. En este caso, el *array a* tiene de tipo la unión de *int* y *str*.

En la siguiente sentencia del ejemplo podemos ver como se añaden nuevos elementos al *array*, en este caso un entero de valor 4.

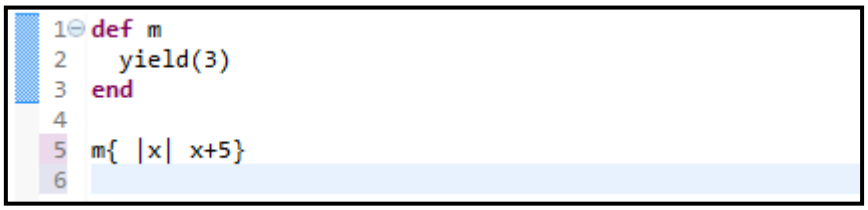


FIGURA 9: EJEMPLO DE BLOQUE EN RUBY

En la **Figura 9** podemos ver un ejemplo de algo realmente característico de Ruby: los **bloques**. En la función *m* tenemos una sentencia *yield*. *Yield* funciona de forma similar a como lo haría un

return, devuelve un valor a una función superior. Sin embargo *yield* es mucho más flexible ya que permite que la función continúe tras ser llamado (al contrario que *return*), y lo que es más importante, se puede usar para inyectar código al programa incluso en tiempo de ejecución.

En nuestro caso, la función *m* pasa al bloque de código el valor 3. Ese 3 se captura en la sentencia "*m{ |x| x+5 }*" y por lo tanto esta función *m* devuelve en este caso el valor 8 al ser invocada. Esto significa que el valor que devuelva el *yield* pasa a ser una variable llamada *x* en este caso, y a esa variable se le suma un 5. Este ejemplo es muy sencillo, pero a continuación vamos a ver otro donde se muestre un poco mejor la verdadera potencia de los bloques de código.

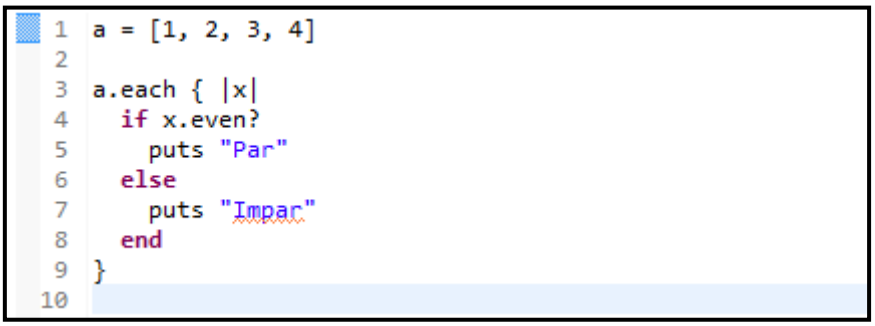


FIGURA 10: OTRO EJEMPLO DE BLOQUES EN RUBY

En la **Figura 10** podemos ver un *array* formado por los números enteros del 1 al 4. Sobre ese *array* aplicamos la función ***each*** que lo que hace es iterar sobre el propio *array* elemento a elemento. En el bloque de código de la función *each*, tenemos que cada elemento del *array* se va guardando en la variable *x*, sobre la cual podemos actuar como queramos. En realidad, la función *each* actúa internamente como un *yield* por cada elemento del *array*. En este ejemplo, comprobamos si es par o no y lo imprime por pantalla, siendo el resultado de ejecutar este código algo como "Impar Par Impar Par", ya que se ejecuta sobre los valores 1, 2, 3 y 4 respectivamente.

Por lo tanto podemos ver que los bloques suponen todo un aliciente para los programadores de Ruby, ya que permiten una gran cantidad de opciones y posibilidades para poder realizar ciertas cosas que en otros lenguajes sería imposible o al menos mucho más complicado.

2 ESTADO DEL ARTE

Ya comentamos anteriormente que el problema del tipado de las variables en los lenguajes de programación dinámicos no es precisamente nuevo y desde su propia creación ha sido un objeto de estudio y un reto para los programadores.

En este apartado vamos a comentar **tres trabajos** realizados anteriormente con tres enfoques distintos para el mismo problema y así poder tener una visión global del mismo.

2.1 TRABAJOS RELACIONADOS

2.1.1 RTC (RUBY TYPE CHECKER)

Ruby Type Checker [5] (**RTC** de ahora en adelante) es un proyecto de la Universidad de Maryland del año 2013. Teniendo en mente el problema del tipado de las variables en Ruby, ellos proponen una **solución intermedia** entre el tipado estático y el dinámico, esto es que los tipos se resuelvan después de cuando lo haría estáticamente pero antes de cuando lo haría dinámicamente, es decir, en tiempo de ejecución pero justo antes de la propia ejecución.

RTC está diseñado como una **librería adicional** a Ruby, de la cual los programadores pueden hacer uso para pseudo-tipar las variables que quieran. No solo eso, RTC también permite hacer **anotaciones** a clases, métodos y objetos, cosa que es común encontrar en los lenguajes estáticos pero no en los lenguajes dinámicos por las razones ya comentadas en puntos anteriores.

Así pues, al usar RTC nos encontraremos con ciertos **métodos** de los que tendremos que hacer uso **para añadir la información de tipado** que consideremos necesaria en nuestro programa, dependiendo claro está de a qué queramos añadir esta información.

En cuanto a funcionamiento interno, las funciones de RTC al ser llamadas lo que hacen es convertir cada elemento en un objeto llamado **Proxy** donde se guarda el propio elemento con su tipo. Este Proxy hace uso del ***method_missing*** característico de Ruby para que cuando se use una función de Proxy primero se asegure de que los argumentos son del tipo correcto, después delega en el propio

objeto que se guardó en el Proxy y finalmente compruebe el tipo del retorno antes de devolvérselo a quien invocó la función.

```
require ' rtc_lib '

class Person
  rtc_annotated
   ...
  typesig "personnel_id : () → Fixnum"
  def personnel_id ... end

  typesig "self . from_id : (Fixnum) →Person"
  def self . from_id(id) ... end

  typesig "manager: () → Manager or %false"
  def manager ... end
end

class Manager < Person
  rtc_annotated
  def employees
    # ... find all managed employees in the database
  end
   ...
  typesig("employees: () → Array<Person>")
end

class Payroll
  rtc_annotated
   ...
  typesig "self . give_raise :(Fixnum,Fixnum,Fixnum)→Fixnum"
  typesig "self . give_raise :(Person,Manager,Fixnum)→Fixnum"
  def self . give_raise (emp, okayed_by, incr)
     ... # ensure okayed_by is in charge of emp
    curr = fetch_salary_from_database (emp)
    set_salary (emp, curr + incr)
  end
end

ids_1 = [1141,1231,3142] # raw, untyped value
ids_1 . push "foo"       # allowed for raw value
ids_2 = [1141,1231,3142]. rtc_annotate ("Array<Fixnum>")
ids_2 . push "foo"       # type error

# Assuming employee number 1141 is a Manager
m = Person.from_id(1141)
m.employees # type error
m_1 = m.rtc_annotate "Manager" # type error
m_2 = m.rtc_cast "Manager" # ok
m_2.employees # ok

sm = m.manager # sm: Manager or %false
unless sm
  ssm_1 = sm.manager # type error
  ssm_2 = sm.rtc_cast("Manager").employees # ok
end
```

FIGURA 11: EJEMPLO DE USO DE RTC

En la **Figura 11** tenemos un extracto de **código anotado con RTC**. Podemos ver casos en concretos como "*typesig "personnel_id: () -> Fixnum""*" en el cual se anota la función *personnel_id* añadiendo la información acerca de sus parámetros de entrada (en este caso ninguno) y su valor de retorno (en este caso un Fixnum).

También podemos ver ejemplos de variables anotadas, en el caso de "*sm = m.manager # sm: Manager or %false*" en la que se aclara que

la variable *sm* debe contener una clase de tipo Manager o si no estaríamos ante un error.

Ya hemos hablado acerca de las ventajas que supone RTC pero también vale la pena hablar sobre sus **inconvenientes**. Para empezar **no es un análisis estático** de código ya que las comprobaciones las realiza en **tiempo de ejecución**. Al ser una librería extra para añadir anotaciones, el código Ruby resultante queda **lleno de sentencias extra** que hace que deje de ser portable y además solo entendible por otros programadores familiarizados con RTC. Para terminar, al añadir comprobaciones adicionales en tiempo de ejecución, **afectará al rendimiento** del programa que en casos de código extenso supondría una diferencia más que notable.

2.1.2 DIAMONDBACK RUBY (DRUBY)

Diamondback Ruby o **DRuby** [6] se presenta como una **extensión al propio lenguaje** que busca subsanar todos aquellos errores causados por la falta de tipado estático de la forma más obvia: añadiendo su **propia especie de tipado estático** a Ruby. De una forma similar a la que vimos en el punto anterior con RTC, DRuby pone a disposición de los programadores ciertos **métodos propios** para solucionar este problema.

En palabras de los propios desarrolladores, DRuby podría ser resumido en **3 puntos clave**: Primero, DRuby ofrece un **lenguaje adicional de tipado** para subsanar las carencias de Ruby que incluye implícitamente los tipos unión e intersección, tipado de objetos, el tipo "self", polimorfismo paramétrico, tipo tupla para arrays heterogéneos y tipos opcionales para los parámetros de funciones. Segundo, DRuby ofrece una **interfaz para añadir anotaciones** a estos tipos y así facilitar futuros trabajos teniendo ya esa parte de documentación ya hecha. Y por último, también incluye un **algoritmo de inferencia básica de tipos** de forma que deja el uso del tipado estático a elección del programador, decida o no usarlo este algoritmo actuará detectando ciertos posibles errores cometidos y advirtiendo de ellos antes de la ejecución.

La parte de la adición de lenguaje para el tipado estático no es la que más interesa de cara a nuestro proyecto, es más bien la parte de la inferencia de tipos que incluye DRuby. Para lograrlo, ya que Ruby permite cierta libertad de sintaxis y por lo tanto no todas las sentencias se escriben igual a pesar de ser del mismo tipo, han

desarrollado su propio parser del lenguaje para trasformar el código en un **AST** (*Ab*str*act Syntax Tree – Árbol de s*int*axis ab*str*acta*) el cual traducen a **RIL** (Ruby Intermediate Languaje – Lenguaje intermedio Ruby) que lo que consigue es **normalizar** todas esas expresiones que comentamos anteriormente y formaliza el flujo de control. Así pues, es mucho más sencillo trabajar con el código en RIL que en la versión original.

En cuanto a la propia inferencia de tipos, DRuby lo hace de forma simple. Primero recorre todo el programa generando **dependencias** de cara a ser resueltas posteriormente. Por ejemplo si encuentra "*x.m()*" asume que *x* es una instancia de una clase que tiene el método *m* y así lo anota en su propio lenguaje estático. Una vez hechas las dependencias, las va verificando consecuentemente para ver si surgen errores. Siguiendo el ejemplo anterior, si tenemos la dependencia que nos dice que *x* es un objeto de clase *A* y dicha clase no tiene ninguna dependencia que indique que tiene un método llamado m, el programa avisará del error.

2.1.3 ECSTATIC

Ecstatic [7] es el proyecto de una tesis de máster de tres alumnos de la universidad danesa de Aalborg, en el año 2007. Se trata de una API desarrollada para **inferir los tipos** de los objetos usados en cierto código Ruby antes de la ejecución. Al contrario que en los dos proyectos anteriores, Ecstatic no propone ningún tipo de lenguaje propio adicional para añadir tipado estático a Ruby, simplemente asigna los tipos sin necesidad de ejecutar el propio código, permitiendo descubrir los errores antes **del tiempo de ejecución**.

Ecstatic tiene **tres módulos** principales que se combinan para realizar el análisis de tipos: **RubySim**, el **Parser** y el **Controller**; que podemos ver en la **Figura 12**:

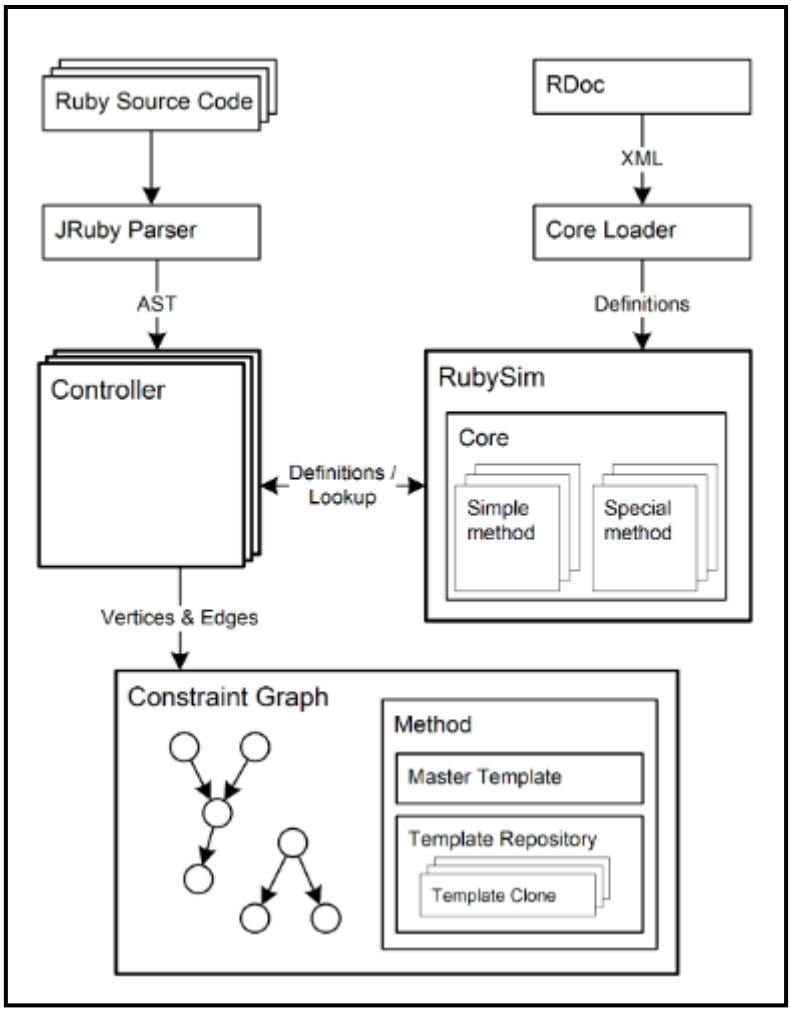


FIGURA 12: GRÁFICO EXPLICATIVO SOBRE EL FUNCIONAMIENTO DE ECSTATIC

Si antes hablábamos de que Ecstatic es capaz de inferir los tipos del código Ruby sin necesidad de ejecutarlo, esto no es del todo cierto. Si bien el código no se ejecuta, **RubySim** es precisamente la abreviatura de Ruby Simulator, un módulo creado en C para **simular la ejecución** del código Ruby que se planea analizar. Para ello, han tenido que programar manualmente una gran cantidad de funciones propias de Ruby adaptándolas al lenguaje C para que la simulación sea lo más fiel posible.

Simulando el comportamiento del código Ruby, lo que se consigue es tener una **vista final** de cómo quedarán todos los objetos del programa y poder realizar grafos necesarios para la inferencia de tipos. Sin embargo esto significa una **penalización en cuanto al rendimiento** y no solo eso, sino que la esencia del tipado estático queda un poco en entredicho ya que en realidad la simulación se podría considerar como tiempo de ejecución.

También hacen uso de un **parser** para pasar del código original de Ruby a un **AST** (Árbol de Sintaxis Abstracta) que sirve para unificar y simplificar el código y hacer de él mucho más fácil de usar, como explicamos en el proyecto anterior.

Una vez tenemos el código normalizado gracias al parser y los grafos hechos gracias al simulador, el **Controller** se encarga de la **inferencia de tipos** en sí. En este caso, Ecstatic hace uso de **CPA** (Cartesian Product Algorithm - Algoritmo de Producto Cartesiano). Este algoritmo es muy sencillo de aplicar y más aun cuando se tiene el código en forma de grafo o árbol como es este caso. Para ilustrarlo lo mejor es ver el ejemplo de la **Figura 13**:

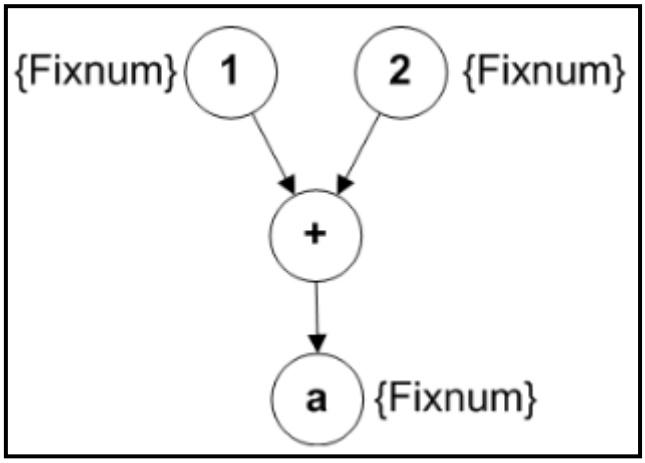


FIGURA 13: GRAFO ILUSTRATIVO DE CPA

Si tenemos una sentencia *"a = 1+2"*, en la figura se puede observar cómo sería el grafo generado a partir de esa sentencia y la lógica que sigue este algoritmo, donde asume que *a* es un *Fixnum* ya que se le asigna el conjunto de los tipos de los nodos 1 y 2 (*Fixnum*).

Una vez hecho el tipado, Ecstatic ofrece al usuario el AST generado a partir del código Ruby y también una versión del propio **código Ruby anotado con los tipos** de cada objeto usado.

2.2 HERRAMIENTAS UTILIZADAS

Un aspecto importante a la hora de trabajar con código de un lenguaje de programación, en este caso Ruby, es disponer de un parser capaz de generar un **AST** (Árbol de Sintaxis Abstracta), como hemos podido observar al estudiar los trabajos descritos anteriormente.

A la hora de elegir un parser se nos planteó la primera gran decisión de cara al futuro del proyecto: en qué lenguaje iba a ser desarrollado. Por un lado teníamos **JParser** [8] sobre el cual tendríamos que trabajar en **Java**, y por otro lado teníamos el **parser de Whitequark** [9] hecho sobre **Ruby**.

En cuanto a funcionalidad, los dos realizaban un trabajo con un resultado similar, así que la elección se basó más en la idea de trabajar en Ruby de manera nativa, ya que el proyecto trata sobre este mismo lenguaje.

El **parser de Whitequark** tiene un funcionamiento muy sencillo: al recibir un código escrito en Ruby realiza el **análisis sintáctico** y devuelve el resultado en forma de AST, siendo ese árbol un conjunto de **nodos anidados**. El parser tiene una gran cantidad de tipos de nodo para poder cubrir toda la complejidad del lenguaje y ofrece ciertos métodos para manipularlo.

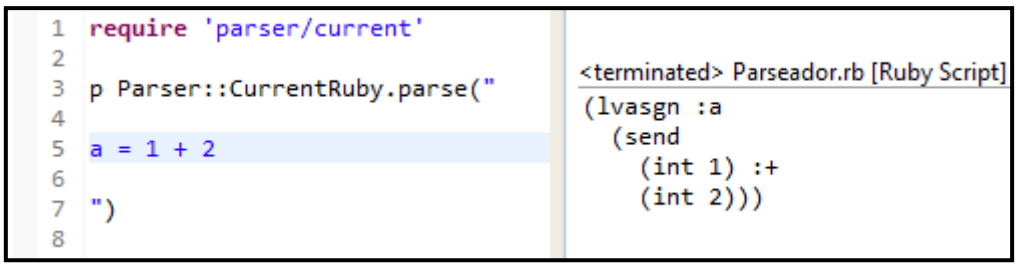


FIGURA 14: EJEMPLO DE FUNCIONAMIENTO DEL PARSER DE WHITEQUARK

En concreto, este parser ofrece una librería llamada AST con dos únicas clases: ***Node***, ***Processor*** y un módulo muy simple llamado *Sexp*. La clase ***Node*** es la más interesante y en la que se basa el proyecto. La estructura básica de un nodo tiene esta forma: *"(int 3)"* aunque existen también nodos más complejos con otros anidados como *"(send (int 1) :+ (int 2))"*. En general, la primera parte indica el **tipo del nodo** (*int*, *send*, etc.) accesible mediante el método *node.type()*. Después, el nodo puede tener símbolos (3, :+, etc.) y otros **nodos anidados**, creando así el árbol. A esos nodos hijos se puede acceder mediante la función *node.children()*.

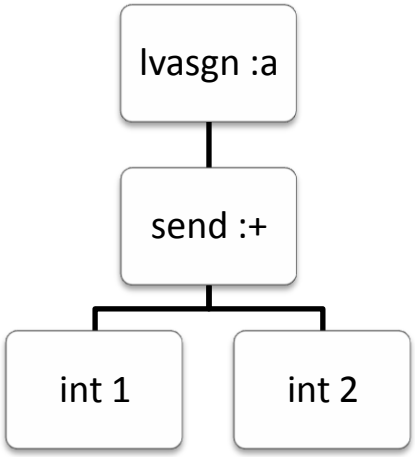


FIGURA 15: ARBOL CONSTRUIDO POR EL PARSER A PARTIR DE CÓDIGO RUBY

En la **Figura 15** ver un ejemplo del árbol formado al parsear la sentencia *"a = 1 + 2"* que nos devolvería los nodos de esta forma: *"(lvasgn :a (send (int 1) :+ (int 2)))"*. También podemos distinguir fácilmente entre los **nodos terminales** que contienen un literal y los **no terminales** que contienen a otro nodo anidado. Veremos más ejemplos de estos árboles y de más tipos de nodo más adelante.

La otra clase que nos interesa del parser es la llamada ***Processor***, que es el pilar clave del proyecto. Esta clase sirve para **controlar los nodos** que devuelve el parser y poder **procesarlos** a nuestro antojo. En especial hablamos del método *process(node)* que lo que hace es enviar el nodo al método adecuado para procesarlo. Esos métodos se tienen que definir en la clase *Processor* por el propio programador de la forma *on_tipoDeNodo(node)*.

Por ejemplo si llamáramos a la función *process(node)* sobre el nodo del ejemplo anterior (de tipo *lvasgn*), el *Processor* mandaría ese nodo a la función *on_lvasgn* (que debe estar implementada por el programador). Si no hay ninguna función para recibir el nodo, se ejecuta el método *handler_missing(node)* que devuelve un *nil*.

Resumiendo, el parser de Whitequark no solo ofrece la posibilidad de **normalizar y traducir** un código Ruby a un AST, sino que también incluye una clase como *Processor* capaz de gestionar y manipular los nodos del árbol para permitir al programador la libertad de usar la información que provee dicho parser.

3 DISEÑO Y DESARROLLO

3.1 NOCIONES BÁSICAS

Nuestro analizador se basa en un concepto sencillo. Visto desde fuera, **recibe un programa** escrito en Ruby y **devuelve una estructura con toda la información** de las variables, clases, métodos etc. utilizados en el programa y por supuesto con sus tipos correspondientes. A esta estructura se le puede preguntar sobre los errores cometidos y sobre el tipo de alguna variable en concreto, aunque también se le puede pedir toda la información completa de los objetos del programa.

Si comprobamos su funcionamiento interno, cuenta con **dos fases** diferenciadas. La primera es el **análisis sintáctico** del código que realiza el parser de Whitequark explicado en el apartado anterior. La segunda fase y principal es el **análisis estático** del AST devuelto por el parser para inferir los tipos de todas las variables del programa e introducir toda la información en la estructura que se devuelve como resultado.

FIGURA 16: DIAGRAMA DE FUNCIONAMIENTO DEL ANALIZADOR

Centrándonos en la fase de análisis estático y más concretamente en la inferencia de tipos, se pueden distinguir **dos tipos de asignaciones** en Ruby: asignaciones a un literal o lo que llamaríamos **nodos terminales** (a = 1), y asignaciones a otro nodo o **conjunto de nodos** (b = a, b = 1 + 2). Siguiendo este razonamiento, para estas últimas asignaciones haremos uso de una lógica similar a la del CPA (Cartesian Product Algorythm – Algoritmo de Producto Cartesiano), esto es **resolver los tipos de los nodos terminales e ir propagándolos** hacia el resto de nodos no terminales.

3.2 DESARROLLO

Una vez vistas las nociones básicas sobre cómo haremos el tipado, en esta sección explicaremos más en detalle los procedimientos que seguimos para conseguir **procesar los distintos tipos de nodos** que encontramos en el AST, dando ejemplos concretos para ilustrar la técnica.

También explicaremos los detalles más generales de la aplicación según vayan surgiendo y al final resumiremos todo para dar una visión más completa de cómo está diseñada. En concreto en esta sección pasaremos a explicar el **funcionamiento interno** del analizador para llegar al objetivo que es devolver una estructura con toda la información acerca de las variables y objetos usados en el programa, así como todos los reportes de errores que se hayan detectado.

Para entender mejor este funcionamiento hay que explicar que en cada código Ruby vamos a distinguir **3 ámbitos de programa** distintos y para ellos cada uno de ellos organizaremos el tipado de forma distinta, así como la manera de incluir la información en la estructura de resultados. Estos 3 ámbitos son: **ámbito general**, **ámbito de función** y **ámbito de clase**.

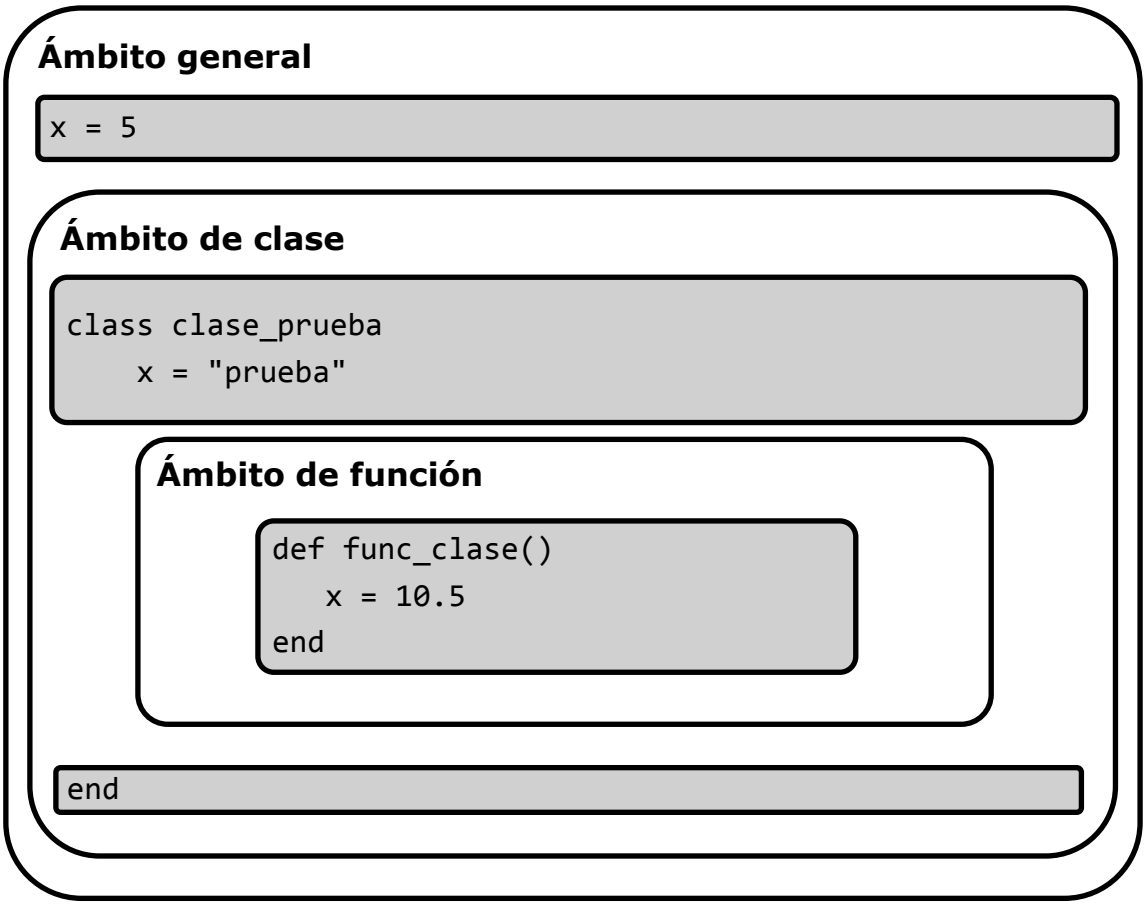


FIGURA 17: EJEMPLO PARA ILUSTRAR LOS DISTINTOS ÁMBITOS DE PROGRAMA

Para hacer esta división nos hemos basado en un **concepto muy sencillo**. En un programa simple no se pueden declarar dos variables distintas con el mismo nombre, salvo que estén en distintos ámbitos de programa. Así pues podría haber una variable *x* dentro del ámbito general que fuera distinta a otra variable *x* que estuviera dentro de una función (ámbito de función) que a su vez fuera distinta a otra variable *x* que estuviera dentro de una clase (ámbito de clase). Este ejemplo se puede ver más claramente en la **Figura 17**.

3.2.1 ÁMBITO GENERAL

En el caso del ámbito general utilizaremos una **tabla *hash*** en la que a cada variable le asignaremos su tipo o un *array* con sus tipos si es que tiene más de uno. En caso de que no tenga un tipo definido o de que haya un error al asignarlo en el programa, en la tabla hash le daremos el tipo *nil*.

Al ser el ámbito general lo que sería el "**main**" en otros lenguajes tradicionales como C, encontraremos que aquí tenemos la mayor

parte del programa y que **engloba a los otros dos ámbitos** que son el de clase y el de función. Así pues vamos a ver en este apartado cómo es el tipado de los elementos básicos del lenguaje Ruby.

3.2.1.1 ASIGNACIONES SIMPLES

En esta sección veremos cómo se comporta el analizador a la hora de procesar **asignaciones** a los distintos tipos de variables. En la **Figura 18** podemos ver 3 tipos distintos de variable. *$a* es una variable global, *b* es una variable local y *@d* es una variable de instancia.

```
1 $a = 1
2 b = 2
3 c = "3"
4 @d = 4.0
5 e = $a + b
6
```

FIGURA 18: EJEMPLO DE ASIGNACIONES EN RUBY

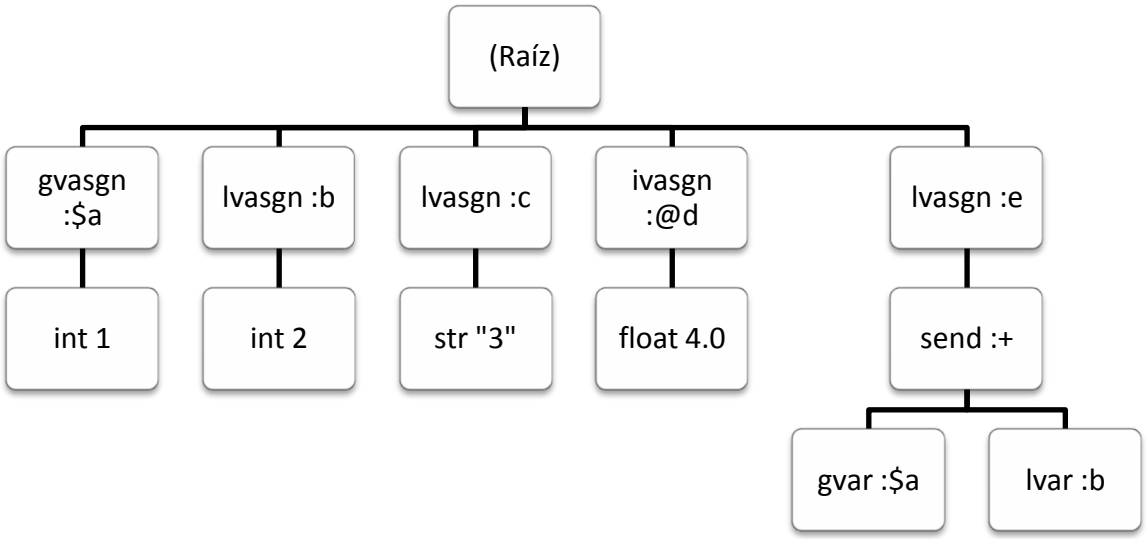


FIGURA 19: AST GENERADO A PARTIR DE LA FIGURA 18

Ya comentamos que al estar en el ámbito general de programa, el tipo de estas variables lo guardaremos en una **tabla *hash***. En el caso de las primeras asignaciones más simples, el analizador simplemente **comprueba los nodos terminales** para obtener el tipo de las variables (*int*, *str*, *float*).

En el último caso del ejemplo, nos encontramos que a la variable e le asignamos la suma de otras dos variables. Para este caso el analizador al recibir el nodo *send* **resuelve** la función (una suma en este caso) **comparando los tipos** de *$a* y *b*, que son compatibles ya que ambos son de tipo *int*, por lo tanto e **hereda** ese mismo tipo. Si

por ejemplo intentáramos sumar *b + c*, el analizador reportaría un error ya que no se pueden sumar *int + str*, y *e* quedaría con el tipo *nil*.

Otro caso distinto sería por ejemplo si a la variable *b* que actualmente es de tipo *int* le intentáramos asignar un valor de tipo *str*. Ya hemos comentado que en Ruby los tipos son dinámicos y por lo tanto permitiría este cambio. En el caso del analizador, reportará un aviso acerca del **cambio de tipo** y la variable *b* quedaría tipada finalmente como *str*.

3.2.1.2 CONDICIONALES Y BUCLES

En este caso vamos a poder ver cómo se comporta el parser en los casos de **condicionales** con *if* y **bucles** con *while*. A pesar de solo tener un bloque agrupado en el caso del *if* en el código, el parser lo divide en nodos separados salvo el *else* que lo incluye junto al último *if*. El caso del bucle *while* es muy similar, primero tiene una rama que resuelve la condición y después la siguiente rama es la sentencia a ejecutar si la condición se cumple.

```
1  if a > 5
2    b = 1
3  elsif a == 5
4    b = 2
5  else
6    b = 3
7  end
8
9  while a > 3
10   b = 4
11 end
12
```

FIGURA 20: EJEMPLO DE CONDICIONALES Y BUCLES EN RUBY

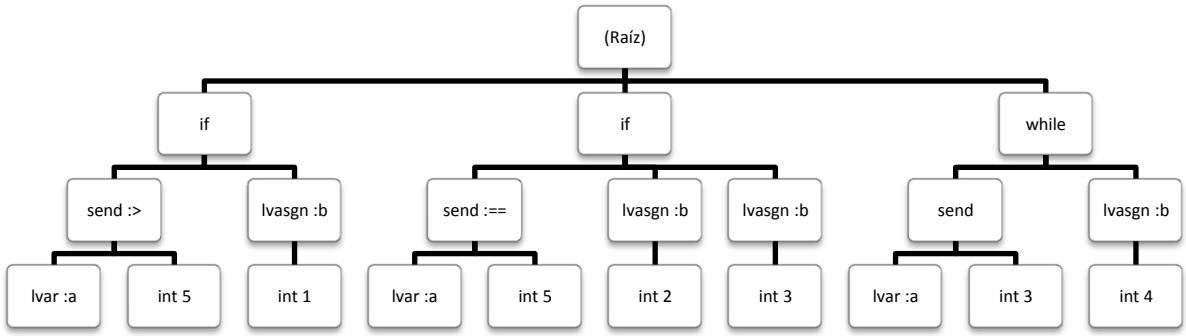


FIGURA 21: AST GENERADO A PARTIR DE LA FIGURA 20

De cara al analizador, tenemos que comprobar por una parte si los tipos de las variables utilizadas como condición son válidos, es decir, si existen esas variables y **si los tipos son compatibles** (en este ejemplo, a tendría que ser de tipo *int*). Por otra parte, tenemos las asignaciones que comentamos en el punto anterior y que no necesitan ningún comentario añadido a esa explicación.

3.2.1.3 ARRAYS Y BLOQUES

En nuestro analizador necesitamos dar a los ***arrays*** un **trato especial**. Ya comentamos anteriormente que los *arrays* funcionan de forma distinta en Ruby que en los lenguajes de programación estáticos. En Ruby no hay que asignarles un tipo específico y por eso pueden **mezclar elementos de distintos tipos**, como en este ejemplo.

```
1 arr = [1, "2", 3.0]
2
```

FIGURA 22: EJEMPLO DE ARRAYS EN RUBY

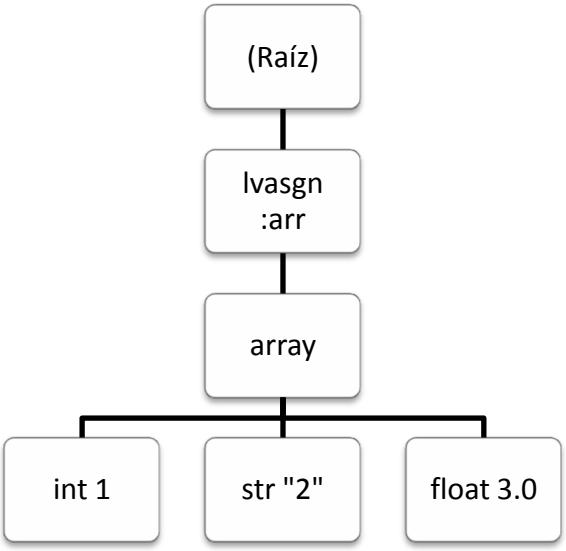


FIGURA 23: AST GENERADO A PARTIR DE LA FIGURA 22

El *array arr* contiene un elemento de tipo *int*, otro de tipo *str* y otro de tipo *float*. En realidad el tipo de *arr* es del **tipo Unión**, una unión de estos tres tipos distintos, pero lo que asignamos en la tabla hash del analizador es un tipo propio llamado *array*. Sin embargo, con esto no es suficiente ya que tenemos que aclarar qué tipos conforman la unión de tipos del *array*.

Para ello, hemos creado una **estructura propia** para almacenar la información relativa a ellos. En la estructura ***ArrayDef*** almacenamos el nombre y un *array* con los tipos de las variables que contiene el array que queremos almacenar. Todas estas estructuras *ArrayDef* usadas para la información de los *arrays* del ámbito general de programa se almacenan en un apartado específico en la estructura de resultados.

3.2.2 ÁMBITO DE FUNCIÓN

Entramos en un nuevo ámbito, en este caso el **ámbito de función**. Así como ocurre en los lenguajes tradicionales, las **variables** declaradas dentro de una función son **independientes** del resto del programa permitiendo así que por ejemplo pueda haber variables con el mismo nombre dentro y fuera de la función.

3.2.2.1 FUNCIONES

Al estar en un nuevo ámbito no podemos seguir almacenando la información del mismo modo que lo hacíamos en el ámbito general ya que en este caso tenemos más datos a tener en cuenta. Este es el motivo por el cual hemos decidido crear una **estructura especial** capaz de almacenar la información relativa a las funciones llamada ***FunctionDef***.

```
1  def Prueba (x, y)
2    return x+y
3  end
4
5  a = Prueba(1, 2)
6
```

FIGURA 24: EJEMPLO DE FUNCIÓN EN RUBY

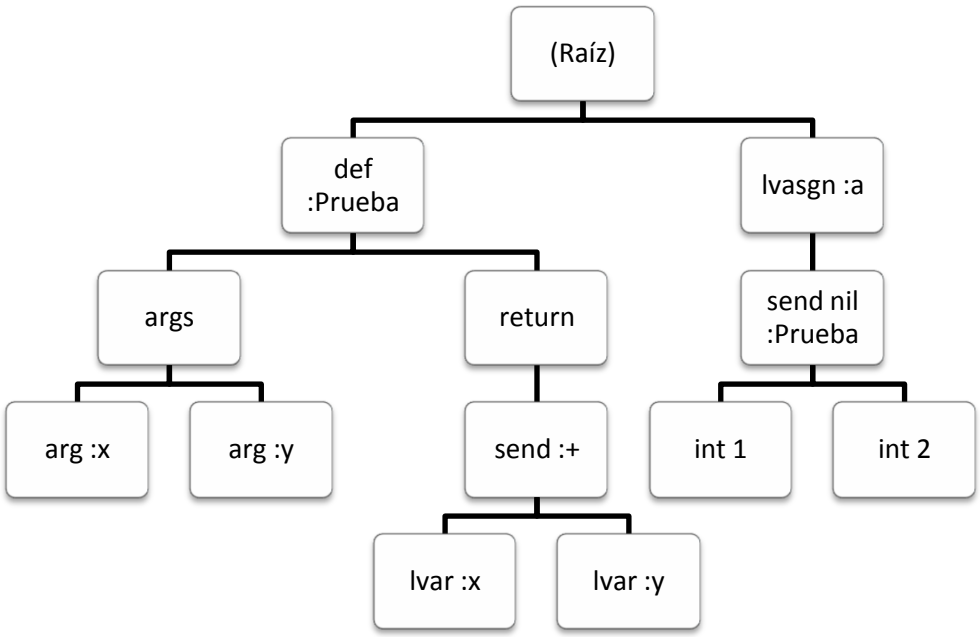


FIGURA 25: AST GENERADO A PARTIR DE LA FIGURA 24

Dicha estructura contiene un campo para guardar el **nombre** de la función, otro para guardar el tipo del ***return*** y del ***yield*** si los hubiera y de forma análoga a como sucedía en el ámbito general, un apartado para las **variables** usadas dentro de la función, otro para los **argumentos** y otro para los ***arrays*** propios.

En el caso de la **Figura 24**, tendríamos a las variables *x* e *y* guardadas como argumentos de la función con tipo *nil* mientras que el resto de campos también quedarían a *nil*, al igual que ocurre con la variable de instancia *a*.

Aunque viendo el ejemplo a simple vista es fácil deducir que al ejecutar el código la variable *a* será de tipo *int* y tendrá el valor 3, no lo es tanto a la hora de realizar la inferencia de tipos. En este caso, hasta que no se procesa la sentencia "*a = Prueba(1, 2)*" no es posible saber qué tipo tendrán los argumentos *x* e *y* de la función y por lo tanto tampoco es posible saber qué tipo de valor retornará y qué tipo tendrá la variable *a*.

Cabría la posibilidad de añadir en la estructura de la función la información de que a *x* e *y* se les asignará un valor de tipo *int* pero no sería correcto ya que más tarde podríamos tener una sentencia "*a = Prueba("b", "c")*" que sería correcta y les estaría pasando unos valores de tipo *str*. La única forma de inferir el tipo de *a* correctamente sería hacer una **simulación de la ejecución** del código Ruby de forma similar a la que vimos en la explicación de Ecstatic, pero consideramos que no es realmente análisis estático ya

que realmente lo infiere en un tiempo de pseudo-ejecución y afectaría al rendimiento.

Sin embargo hay otros casos similares en los que sí somos capaces de inferir todos los tipos:

```
1  def Prueba (x, y)
2    b = 1 + 2
3    y = y + 3
4    return x+b
5  end
6
7  a = Prueba(1, 2)
8
```

FIGURA 26: OTRO EJEMPLO DE FUNCIONES EN RUBY

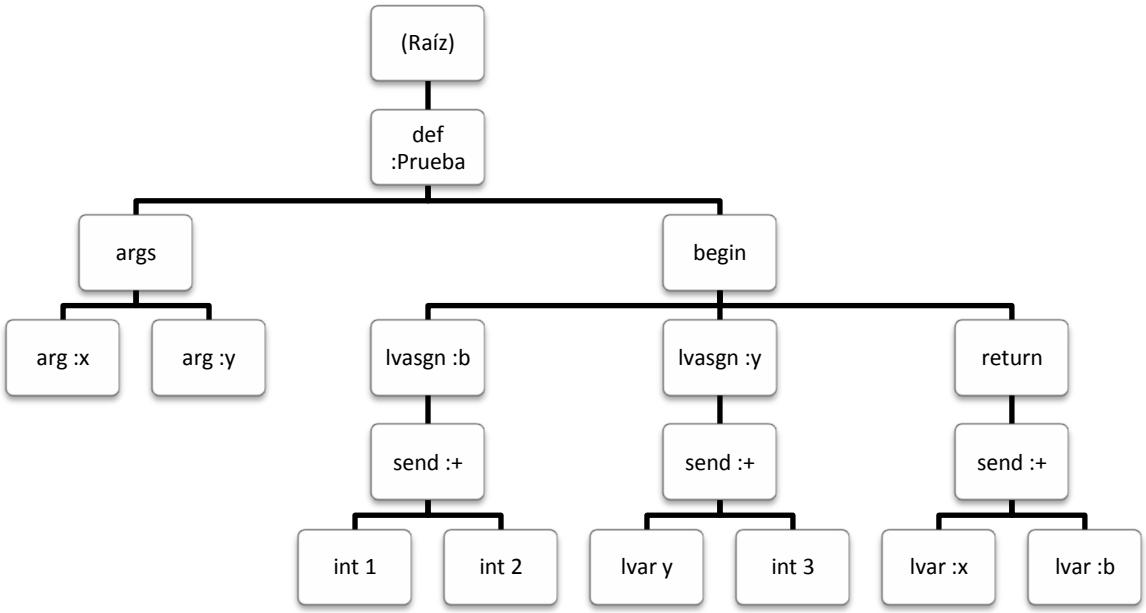


FIGURA 27: AST GENERADO A PARTIR DE LA FIGURA 26 OMITIENDO LA ULTIMA SENTENCIA

En este caso tenemos **varios elementos distintos** respecto al ejemplo anterior. Para empezar tenemos una variable nueva dentro del ámbito de la función llamada *b* que tiene de tipo *int* ya que es la suma de otros dos *int* (1+2). Después tenemos que al argumento *y* se le asigna el valor que ya tuviera más el *int* 3, lo cual nos indica que para que no haya errores el argumento *y* debe ser también de un tipo compatible con *int* (esto es *int* o *float*).

Cabe destacar en este ejemplo que si nos fijamos en el árbol, el parser no establece ninguna distinción entre las variables de función y los argumentos (*b* e *y*) asignándoles el mismo tipo de nodo (*lvasgn* en este caso). Para distinguirlos debemos hacer uso de la información dada en la rama con el nodo *args*.

Siguiendo con el ejemplo, nos fijamos que en el *return* tenemos la suma del argumento *x* con la variable *b* a la que ya le hemos asignado el tipo *int*. Siguiendo la lógica que utilizamos previamente, *x* debe ser de un tipo compatible con *int* (*float* o *int*) y por lo tanto se le asigna ese tipo y lo que es más importante, también **sabemos que la función devuelve un valor de ese mismo tipo**.

Así pues al ejecutar la sentencia "*a = Prueba(1, 2)*" ya sabemos que *a* recibirá el tipo *int*. Este caso es muy concreto ya que el tipo del *return* sabemos que siempre será el mismo y lo hemos podido **inferir solo analizando la función**, pero es sin duda un paso muy importante el ser capaz de "predecir" el resultado de una función sin llegar realmente a ejecutarla. En la **Figura 28** se puede ver un ejemplo del resultado del análisis de este programa en nuestro analizador.

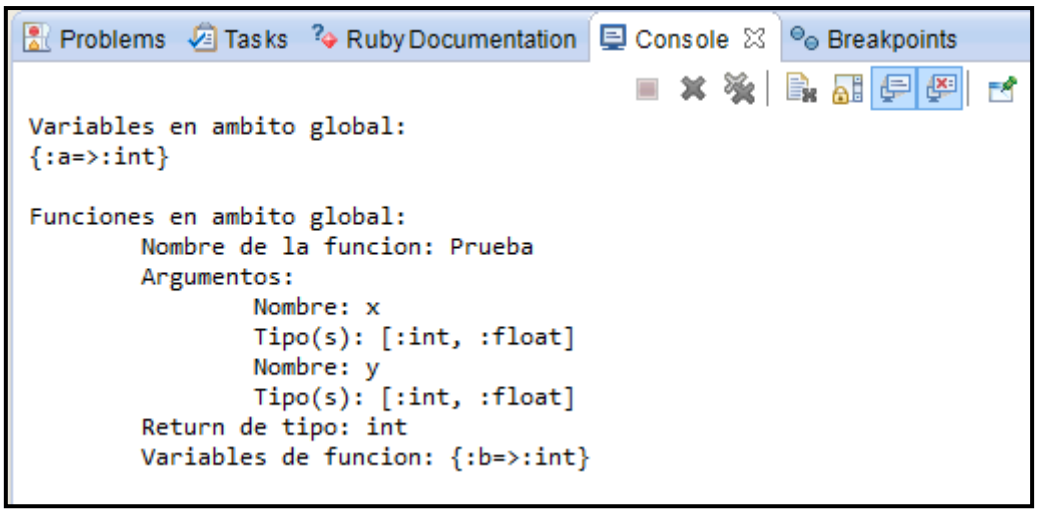


FIGURA 28: RESULTADO DEL ANALIZADOR SOBRE LA FIGURA 26

3.2.2.2 BLOQUES

En la **Figura 29** podemos observar el funcionamiento de los **bloques** en Ruby. Por un lado tenemos la función *Prueba* que al igual que como ocurría en el ejemplo anterior, podemos inferir que tiene un *yield* de tipo *int*. Esa información nos sirve a la hora de analizar el bloque, ya que en este caso tenemos una operación que podría resultar en error.

```ruby
def Prueba (x)
  yield(x+3)
end

Prueba(2){ |var| var+5 }
```

FIGURA 29: EJEMPLO DE BLOQUES EN RUBY

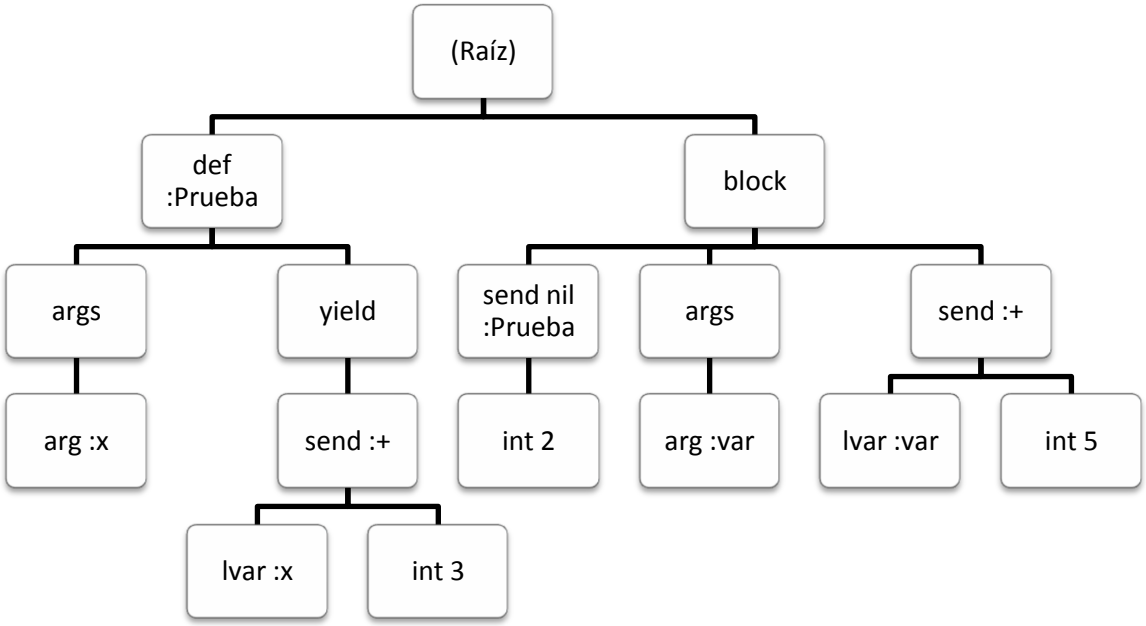


FIGURA 30: AST GENERADO A PARTIR DE LA FIGURA 29

El valor del *yield* que se pasa al bloque en este caso es de tipo *int*, por lo tanto tenemos que **comprobar** que **dentro del bloque** la variable *var* se trata como tal. En este caso basta con comprobar que la operación "*var + 5*" es correcta ya que **ambos tipos son compatibles**.

Merece la pena ver cómo en el árbol **la función y el bloque van separados** al contrario de lo que ocurre en el propio código Ruby. Aun así en el nodo *block* hay una rama para indicar la función correspondiente y los parámetros que recibe dicha función.

Aun así, al igual que ocurría con los *return* del ejemplo anterior, el tipado del *yield* es **circunstancial** y por lo tanto no siempre podemos inferir su tipo, esto significa que a la hora de analizar los bloques que tengan relación con esos *yield* no podremos inferir todos los tipos de las variables involucradas.

Como dijimos en el punto anterior, la única solución sería hacer una simulación de la ejecución del programa pero dejaría de ser análisis completamente estático y consumiría más recursos. Además, una de las particularidades de Ruby es precisamente la inyección de código

en tiempo de ejecución precisamente gracias a los bloques por lo tanto se añadiría una dificultad más elevada aun el tratar de inferir completamente todos los tipos relacionados con los bloques.

3.2.3 ÁMBITO DE CLASES

Finalmente llegamos al último ámbito de programa, en este caso el de clases. De la misma manera que ocurría con las funciones las cuales podían tener sus variables propias independientes del ámbito general, las clases también pueden tener no solo variables propias sino también funciones solo accesibles mediante la clase en cuestión.

3.2.3.1 CLASES

Con las **clases** entramos en el último ámbito de variables que es precisamente el ámbito de clase. Al igual que las funciones podían tener sus **variables propias**, las clases no solo pueden tener también las suyas propias, sino que también pueden tener sus **propias funciones**, haciendo que el ámbito de clase sea muy **parecido al ámbito general**.

En nuestro analizador por lo tanto, para poder almacenar las clases hemos creado una **estructura propia** (de forma similar al caso de las funciones) llamada ***ClassDef*** en la que almacenamos el **nombre**, las **variables de instancia** y las **funciones** que pueda contener.

```
1  class Punto
2    def initialize (x, y)
3      @x = x
4      @y = y
5      a = 10
6    end
7
8    def getx
9      return @x
10   end
11
12 end
13
14 p1 = Punto.new(10, 20)
15
```

FIGURA 31: EJEMPLO DE CLASE EN RUBY

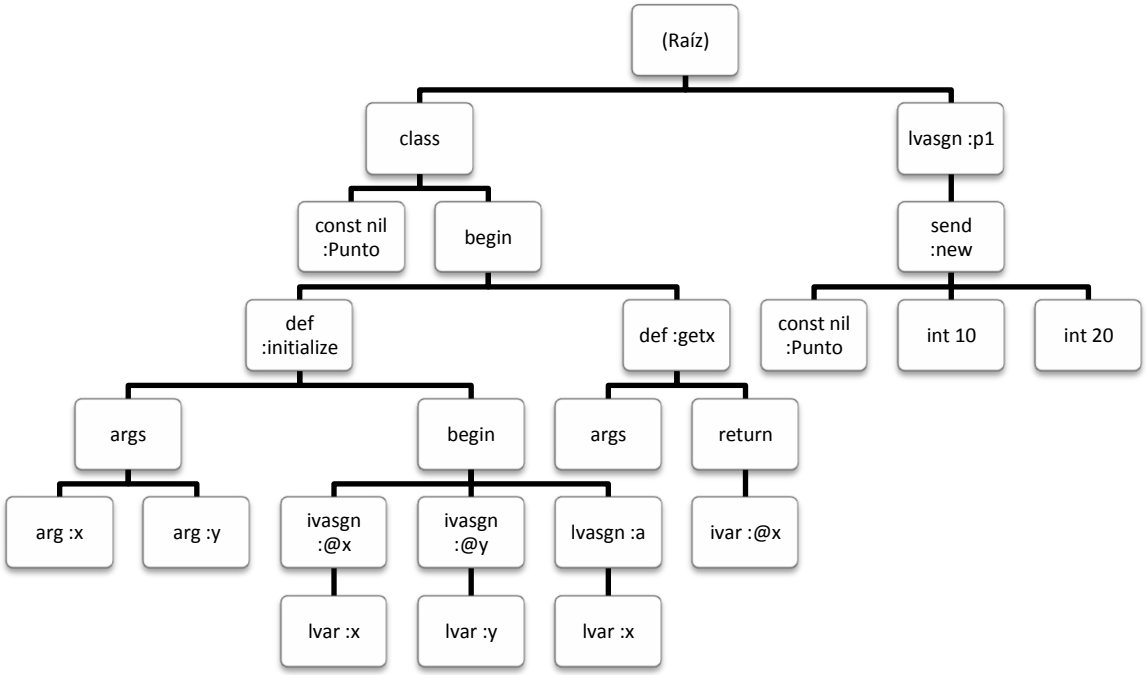
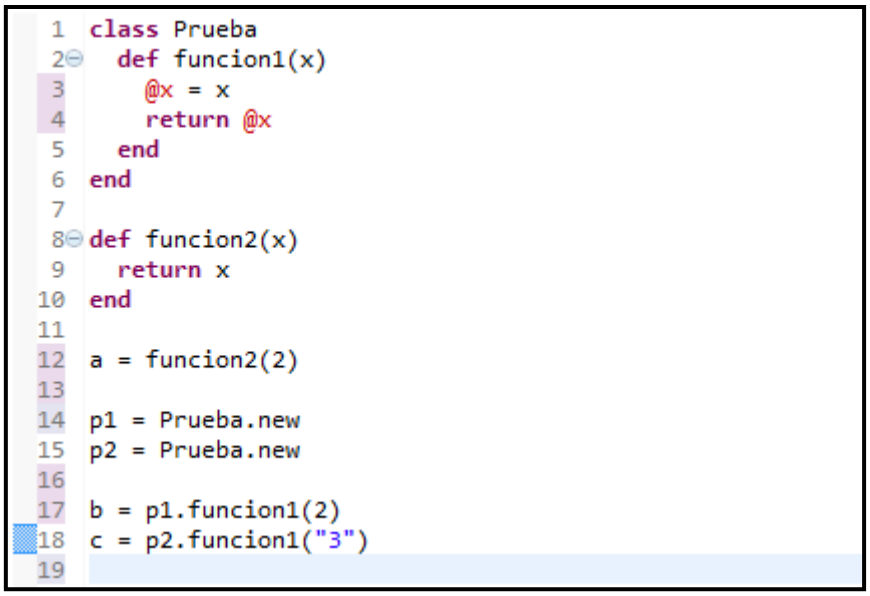


FIGURA 32: AST GENERADO A PARTIR DE LA FIGURA 31

La sintaxis del AST respecto a las clases no es demasiado difícil de entender. Un nodo de tipo *`class`* tiene que tener un nodo hijo de tipo *`const`* que incluya el nombre de la clase, y después el resto viene incluido en un nodo hijo de tipo `begin`. En este ejemplo podemos ver que la clase *`Punto`* tiene dos funciones, cada una con la sintaxis que ya vimos en el apartado de función. También podemos ver un ejemplo de uso de **variables de instancia** con sus nodos propios (*`ivar`*, *`ivasgn`*) y también un ejemplo de **instanciación de la clase**, en el que se forma un nodo de tipo *send* con la información sobre el nombre de la clase que queremos instanciar y los parámetros necesarios.

En este apartado merece la pena señalar y explicar con más profundidad el tratamiento que hace el analizador a la hora de **inferir los tipos relacionados con las clases**. En cierto modo, hay casos en los que nos ocurre el mismo problema que con las funciones, y es que hay tipos que no se pueden inferir hasta que sea llamada más adelante aunque hay varias diferencias. Para ilustrarlo mejor vamos a usar un ejemplo sencillo:



```ruby
class Prueba
  def funcion1(x)
    @x = x
    return @x
  end
end

def funcion2(x)
  return x
end

a = funcion2(2)

p1 = Prueba.new
p2 = Prueba.new

b = p1.funcion1(2)
c = p2.funcion1("3")

```

FIGURA 33: OTRO EJEMPLO DE CLASE EN RUBY

Como se puede ver en la **Figura 33**, tenemos dos funciones que operan de forma muy similar, devolviendo exactamente el mismo valor que se le pasa por argumento. Las **diferencias** están en que en el caso de *funcion1* está **dentro de una clase** llamada *Prueba* y el valor del argumento antes de ser devuelto lo almacena en la variable de instancia *@x*, mientras que *funcion2* está en **ámbito general** y devuelve el argumento directamente.

Siguiendo la lógica a simple vista es fácil deducir que *a* y *b* serán de tipo *int* con valor 2 y *c* será de valor *str* con valor "3", sin embargo no es tan fácil de deducir analizando el AST. Ya vimos en el apartado anterior el problema que teníamos a la hora de analizar los tipos dentro de las funciones, en este caso el tipo del argumento *x* de *funcion2* y por consecuencia el de su *return* y la variable *b*. Es fácil ver que el argumento *x* tiene un tipo **completamente variable** y que ese tipo se heredará a *b*, no es correcto asumir que siempre será *int*.

Esto no es igual en el caso de funcion1 ya que se encuentra dentro del ámbito de clase. En estos casos el analizador actúa de la siguiente manera: Al empezar a procesar una clase, se almacena toda la información de dicha clase en una estructura de tipo *ClassDef* como dijimos anteriormente y todas estas estructuras se almacenan a su vez en un campo del ámbito general llamado simplemente "clases".

Al crear la estructura *ClassDef* no se pueden asignar todos los tipos de las variables de instancia ya que precisamente aun no ha habido instanciación de la clase. Para anticiparnos a lo que vendrá después,

este tipo de variables tienen un campo llamado "**conexiones**" en las que a forma de grafo asignaremos las **dependencias** entre ellas. Si vemos el ejemplo de arriba, la variable *@x* tiene una "conexión" con el argumento *x* de *funcion1*, y a su vez el *return* de esa función tiene otra "conexión" con la propia variable *@x*.

A la hora de procesar por ejemplo la sentencia "*b = p1.funcion1(2)*", el tipo *int* de ese 2 se propagará primero al argumento *x* de *funcion1*, después a *@x*, de ahí al *return* de la función y por último a la variable *b* que será del tipo *int*. Algo similar ocurre al procesar la sentencia "*c = p2.funcion1("3")*", solo que esta vez el tipo que se propaga es *str*.

¿Cómo se consigue entonces que desde la misma clase almacenada como *ClassDef* se admitan dos tipos distintos para las mismas variables? Esto es porque tanto p1 como p2 son dos **instancias distintas de la misma clase** Prueba. Así pues cuando se crea una nueva instancia de clase (en las sentencias "*p1 = Prueba.new*" y "*p2 = Prueba.new*") se hace una copia completa de la información general de la clase contenida en *ClassDef* para después ser capaz de editarla **independientemente de la clase general**.

Es por eso que en la instancia *p1* se propaga el tipo *int* mientras que en la instancia *p2* se propaga el tipo *str*. Y es por esto que en el caso de la función *funcion2* no se puede propagar ningún tipo ya que cada vez que se invoca una función no se crea una instancia de la misma. Si en este ejemplo a la variable *x* de *funcion2* se le asignara el tipo *int*, cuando la misma función fuera invocada con un parámetro de otro tipo distinto produciría un error.

Todas estas **instancias** de clases se almacenan en un **apartado distinto** dentro del ámbito general para que se puedan diferenciar más fácilmente a la hora también de devolver los resultados del análisis.

Vamos a ver un último caso que también merece la pena ser explicado, en la **Figura 34**:

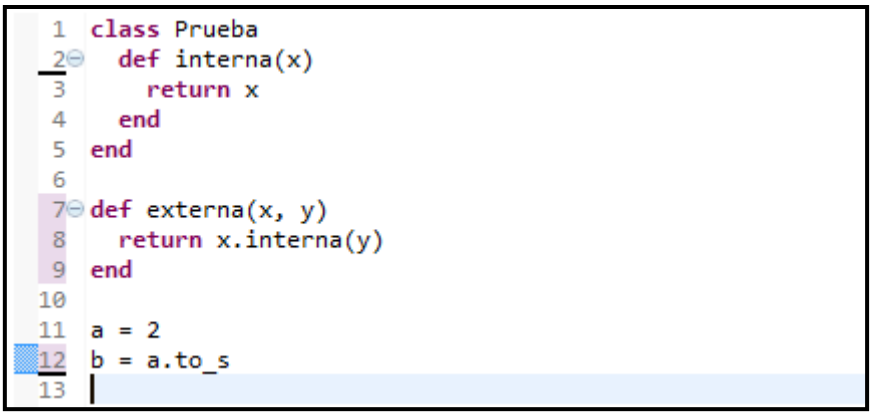


FIGURA 34: EJEMPLO DE CLASES Y FUNCIONES EN RUBY

Siguiendo la filosofía *Duck Typing* propia de Ruby, es fácil ver a simple vista que el argumento *x* de la función *externa(x, y)* tiene que ser de tipo *Prueba* para poder ejecutar sobre él la función *interna(x)*. Sin embargo no es prudente asumir que este argumento *x* será siempre de tipo *Prueba* ya que puede haber más clases que tengan una función *interna(x)* y que también sean capaces de encajar.

El propósito del analizador es asignar **tipos concretos y seguros**, y no tipos posibles, por lo tanto en este ejemplo la variable *x* de la función *externa(x, y)* se quedará sin un tipo asignado hasta que en el código se encuentre una llamada a la función en la cual el parámetro *x* sea un objeto ya tipado.

No solo eso, también podemos ver en la última sentencia *"b = a.to_s"* que se hace uso de una **función propia de las librerías estándar de Ruby** (*to_s*). En concreto esta función se puede ejecutar sobre cualquier objeto en Ruby y siempre devuelve un *string*, con lo cual la variable b sería de tipo *str*. Sin embargo nuestro analizador **no conoce todas las funciones propias** del lenguaje y por lo tanto tampoco es capaz de asignar los tipos relacionados con todas ellas.

Para que el analizador fuera capaz de procesar todas estas funciones deberíamos **añadirle la información de las innumerables funciones de las librerías de Ruby** ya sea introduciéndolas en el analizador del mismo tipo que se introduce la información de las funciones que se declaran en el código o creando una aplicación que hiciera de intermediaria entre las librerías de Ruby y el analizador para realizar el intercambio de información.

Como ejemplo para demostrar que realmente no es un trabajo difícil en cuanto a codificación sino en cuanto a carga de trabajo, en e+l analizador **hemos implementado algunas de ellas** como *to_s* o *to_sym*.

3.3 RESUMEN Y VISTA GENERAL

Así pues ya hemos explicado cómo funciona el analizador con los casos más comunes del lenguaje, así que ahora queda explicar finalmente **la estructura global de la API** y cómo organiza los resultados. Para ello debemos explicar dos últimas estructuras creadas para almacenar toda la información referente al analizador:

La primera es una estructura muy sencilla llamada ***ErrorDef*** en la cual se guardan los **errores y *warnings*** detectados en el código y el lugar donde se producen (línea y columna).

La estructura principal donde se almacenan todos los **resultados** del análisis y que devuelve nuestro analizador es una estructura llamada ***ResultsDef***, la cual tiene además ciertas funciones para acceder a la información.

En concreto tiene 3 métodos para acceder a la **información general** del programa:

- ***daInfoObjetos***: Este método da la información de los objetos creados en el código, esto es: variables de ámbito general, funciones y clases sin instanciar.
- ***daInfoTipos***: Este método pretende dar la información sobre las variables del programa y sus tipos, por lo tanto muestra las variables de ámbito general y las clases instanciadas.
- ***daInfoTotal***: En este caso se muestra la información de las otras dos funciones combinadas.

Adicionalmente, esta estructura también tiene un método llamado ***daErrores*** el cual muestra todos los **errores y *warnings*** cometidos si es que los hubiera.

También incluye dos métodos para acceder a la **información del tipado** de una variable en concreto, que son:

- ***tipoVariable?(var)*** : Busca en todos los campos del programa la variable *var* y devuelve su tipo. En caso de que haya varias variables con el mismo nombre, elige la primera que encuentre en la búsqueda.

- ***tipoVariableEn?(var, lugar)*** : Busca una variable de nombre *var* que este contenida en un objeto de nombre *lugar*, ya sea función o clase y devuelve su tipo. Si el argumento *lugar* está vacío, se buscará la variable en el ámbito global.

```
1  require 'parser/current'
2  require 'Procesador.rb'
3
4  res = procesa("Codigo.rb")
5  res.daInfoTotal
6  res.imprimirErrores
7  res.tipoVariableEn?("x", "funcion2")
8
```

FIGURA 35: EJEMPLO DE USO DEL ANALIZADOR

En la **Figura 35** podemos ver un **ejemplo de uso** del analizador. La función procesa es la que realiza en realidad todo el trabajo. Recibe la ruta del fichero de código que queremos analizar y devuelve la estructura *ResultadosDef* con toda la información. Vamos a analizar internamente su funcionamiento:

```
1  def procesa(fichero)
2    file = File.open(fichero, "rb")
3    contents = file.read
4    file.close
5    arbol = Parser::CurrentRuby.parse(contents)
6    res = ResultadosDef.new
7    prc = RubyAnalyzer.new(res)
8    prc.process(arbol)
9    prc.resolver()
10   res.setArbol(prc)
11   return res
12 end
13
```

FIGURA 36: FUNCIONAMIENTO INTERNO DEL MÉTODO PROCESA

Como podemos ver en la **Figura 36**, lo primero que hace es abrir, leer el fichero donde se encuentra el código y cerrarlo. Mas tarde utiliza el parser de Withequark para crear el **AST** a partir del código (lo que llamamos **análisis sintáctico**). Después creamos la estructura *ResultadosDef* donde almacenaremos todos los resultados del análisis y se la pasamos como argumento al crear la clase *RubyAnalyzer* ya que será el encargado de ello.

El siguiente paso es simplemente **llamar a la función *process()*** de *RubyAnalyzer*, heredada de la clase *Processor* que ya incluía el parser y que se dedica a hacer todas las llamadas a las funciones *on_tipoDeNodo()* dependiendo del tipo de nodo que estemos procesando.

Una vez procesado todo el AST, se llama a la función *resolver()* para que **propague los tipos** que hayan quedado pendientes a través de los campos *Conexiones* que ya explicamos en los apartados anteriores. Una vez hecho esto se añaden todas las estructuras y datos a nuestro *ResultadosDef* para finalmente devolverlo y dejarlo a disposición de los usuarios.

Se puede ver la organización de todas las estructuras creadas para el analizador en el diagrama de clases:

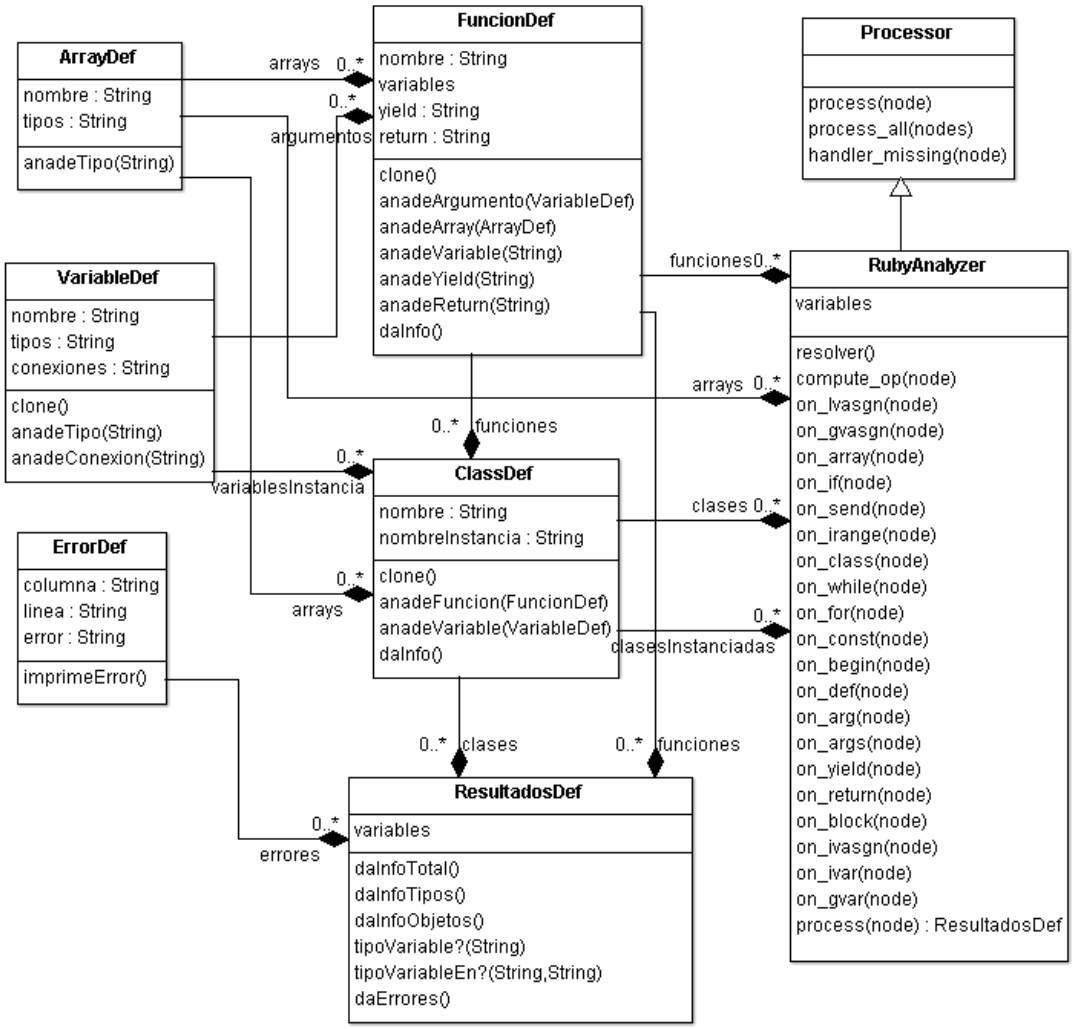


FIGURA 37: DIAGRAMA DE CLASES DEL ANALIZADOR

4 PRUEBAS Y RESULTADOS

En este apartado vamos a comprobar cómo se comporta nuestro analizador frente a programas escritos en Ruby ya existentes y por supuesto a mostrar los resultados del analizador y ver si son correctos o si se presentan nuevos inconvenientes.

A lo largo del desarrollo del analizador se fueron comprobando los elementos más básicos del lenguaje a medida que implementábamos su análisis por lo tanto ya sabemos que los casos más simples funcionan correctamente. El objetivo de estas pruebas es comprobar casos más complejos.

4.1 PRUEBA 1

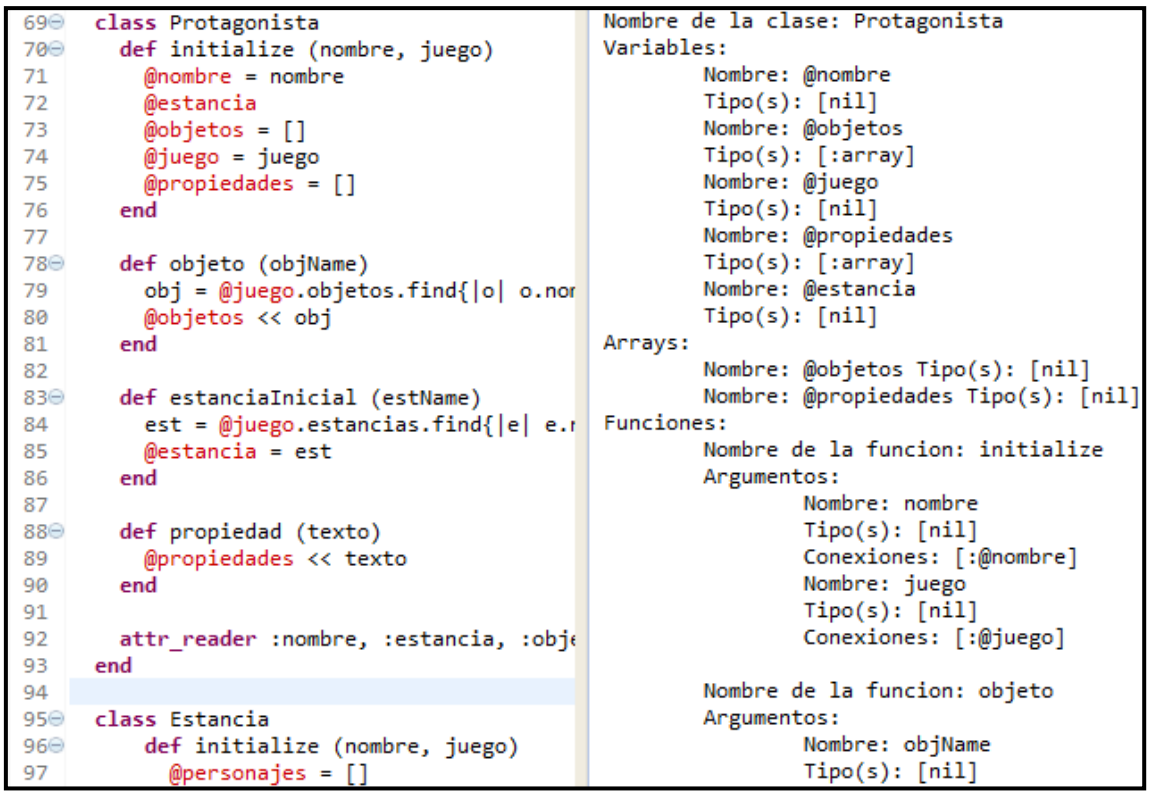
Para esta primera prueba decidimos ejecutar el analizador sobre el proyecto final de la asignatura Desarrollo Automatizado de Software (DASOFT) cursada el año pasado. El proyecto final consistía en una **aplicación desarrollada en Ruby capaz de crear y gestionar videojuegos** del género aventura gráfica. En concreto hemos realizado el análisis sobre el fichero que contenía **todas las clases** del proyecto.

Nuestro analizador solo es capaz de procesar ficheros individuales por lo tanto decidimos analizar solo ese fichero. El proyecto constaba de otro dichero con funciones que hacían uso de esas clases, podríamos haberlos juntado en uno solo para así analizar el proyecto completo pero este segundo fichero contenía bastantes llamadas a métodos *self y eval* que nuestro analizador no es capaz de procesar (esto lo analizaremos más a fondo en la sección de limitaciones) por lo tanto decidimos no incluirlo para no desvirtualizar los resultados.

Como ya hemos explicado, se trata de un **fichero con las definiciones de las clases** de las cuales hará uso el resto del proyecto por lo tanto hay pocas asignaciones concretas. Esta prueba sin embargo es útil para comprobar cómo el analizador es capaz de **sintetizar y almacenar la información** de dichas clases en las estructuras propias creadas para ello.

Con esta prueba nos percatamos de una limitación que no habíamos tenido en cuenta y es que aunque Ruby permita crear instancias de una clase que será declarada más adelante, nuestro analizador no lo

puede asimilar ya que **procesa el código de forma secuencial**. En el caso de este fichero nos encontramos con un ejemplo de este tipo y por lo tanto para poder solucionarlo tuvimos que reorganizarlo de forma que esto no ocurriera. Aun así el analizador no dejaba de funcionar por ello, simplemente reportaba el error de que dichas clases no existían.



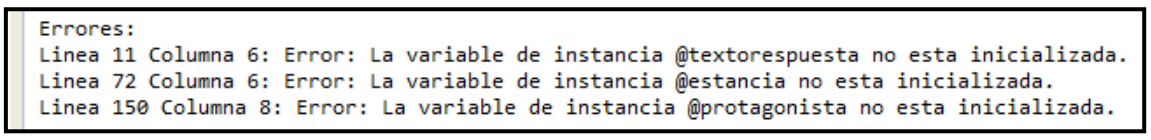
```
class Protagonista
  def initialize (nombre, juego)
    @nombre = nombre
    @estancia
    @objetos = []
    @juego = juego
    @propiedades = []
  end

  def objeto (objName)
    obj = @juego.objetos.find{|o| o.nor
    @objetos << obj
  end

  def estanciaInicial (estName)
    est = @juego.estancias.find{|e| e.
    @estancia = est
  end

  def propiedad (texto)
    @propiedades << texto
  end

  attr_reader :nombre, :estancia, :obj
end

class Estancia
    def initialize (nombre, juego)
      @personajes = []
```

```
Nombre de la clase: Protagonista
Variables:
        Nombre: @nombre
        Tipo(s): [nil]
        Nombre: @objetos
        Tipo(s): [:array]
        Nombre: @juego
        Tipo(s): [nil]
        Nombre: @propiedades
        Tipo(s): [:array]
        Nombre: @estancia
        Tipo(s): [nil]
Arrays:
        Nombre: @objetos Tipo(s): [nil]
        Nombre: @propiedades Tipo(s): [nil]
Funciones:
        Nombre de la funcion: initialize
        Argumentos:
                Nombre: nombre
                Tipo(s): [nil]
                Conexiones: [:@nombre]
                Nombre: juego
                Tipo(s): [nil]
                Conexiones: [:@juego]

        Nombre de la funcion: objeto
        Argumentos:
                Nombre: objName
                Tipo(s): [nil]
```

FIGURA 38: EJEMPLO DE FUNCIONAMIENTO DEL ANALIZADOR EN LA PRUEBA 1

En la **Figura 38** podemos ver un ejemplo de cómo ha procesado el analizador la primera parte de la clase *Protagonista*. Las variables de instancia están todas a *nil* ya que como comentamos se trata únicamente de un fichero de definición de clases y por lo tanto no hay ninguna asignación. Sin embargo podemos ver cómo se han realizado las **conexiones** entre esas variables con los argumentos de la función *initialize* para a la hora de ser llamada se pueda propagar la información de los tipos de dichos argumentos.

En cuanto a los errores encontrados, tenemos el siguiente resultado:



```
Errores:
Linea 11 Columna 6: Error: La variable de instancia @textorespuesta no esta inicializada.
Linea 72 Columna 6: Error: La variable de instancia @estancia no esta inicializada.
Linea 150 Columna 8: Error: La variable de instancia @protagonista no esta inicializada.
```

FIGURA 39: ERRORES ENCONTRADOS EN LA PRUEBA 1

Si nos fijamos en los resultados anteriores, podemos comprobar que efectivamente, en la línea 72 la variable de instancia *@estancia* no está inicializada. Observando el resto de código a simple vista, no se encuentran más errores aparte de los que marca nuestro analizador, con lo cual concluimos que es una **prueba satisfactoria**.

4.2 PRUEBA 2

Para esta segunda prueba hemos usado un fichero que no solo tuviera **definiciones de clase**, sino que también **hiciera uso de ellas**. Para ello tenemos un programa sencillo en el cual se crean unas clases llamadas *Punto* y *Recta* y se realizan varios cálculos en base a ellas.

```
1  class Punto
2    def initialize (x, y)
3      @x = x
4      @y = y
5    end
6
7    def suma
8      puts @x + @y
9    end
10
11   def getx
12     return @x
13   end
14
15   def gety
16     return @y
17   end
18
19   attr_reader :y
20   attr_writer :y
21 end
22
23 class Recta
24   def initialize (x, y)
25     @x = x
26     @y = y
27   end
28
29   def longitud
```

```
Clases:
Nombre de la clase: Punto
Variables:
        Nombre: @x
        Tipo(s): [:int, :float]
        Nombre: @y
        Tipo(s): [:int, :float]
Arrays:
Funciones:
        Nombre de la funcion: initialize
        Argumentos:
                Nombre: x
                Tipo(s): [nil]
                Conexiones: [:@x]
                Nombre: y
                Tipo(s): [nil]
                Conexiones: [:@y]

        Nombre de la funcion: suma
        Argumentos:

        Nombre de la funcion: getx
        Argumentos:
        Return de tipo: [:int, :float]

        Nombre de la funcion: gety
        Argumentos:
        Return de tipo: [:int, :float]
```

FIGURA 40: EJEMPLO DE FUNCIONAMIENTO DEL ANALIZADOR EN LA PRUEBA 2

Podemos ver en la **Figura 40** cómo se ha analizado la clase Punto con sus respectivas funciones. El resultado del analizador corresponde a la **clase en general** y no a ninguna de sus instanciaciones, pero este caso es significativo porque podemos ver que ya tiene asignados algunos tipos. Esto ocurre gracias a la función

suma, que restringe los tipos de *@x* y *@y* a *int* o *float*. También se pueden ver las conexiones que se forman de cara a la posterior **propagación de tipos** a la hora de instanciar la clase.

Para mostrar mejor cómo funciona la asignación de tipos mediante la instanciación vamos a eliminar la función *suma* que veíamos en la **Figura 40**.

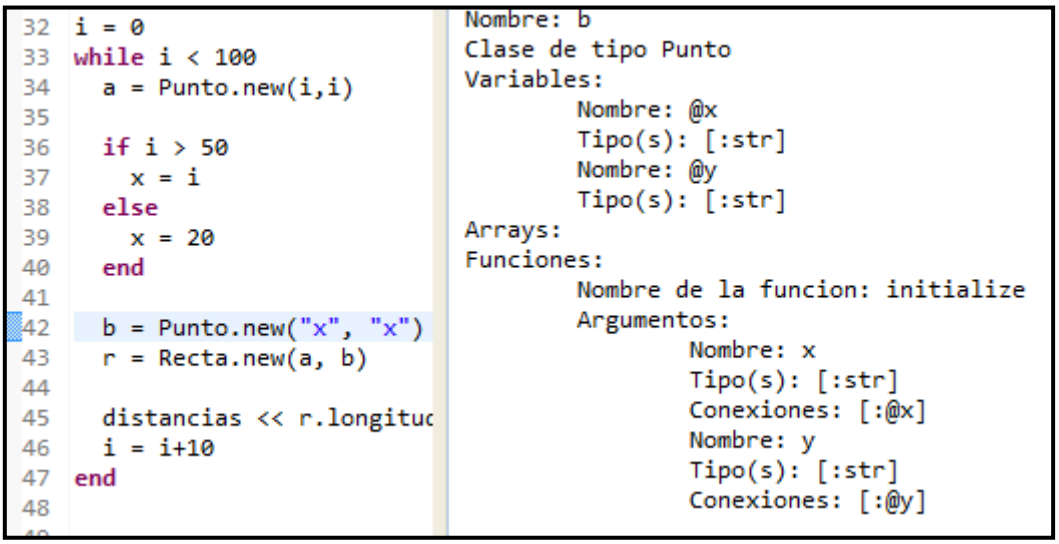


FIGURA 41: OTRO EJEMPLO DE FUNCIONCIONAMIENTO DEL ANALIZADOR EN LA PRUEBA 2

Podemos ver en este ejemplo cómo en la instancia de la clase *Punto* guardada en *b*, las variables de instancia *@x* y @y quedan tipadas como *str*. Esto es porque reciben como parámetro dos *str* y ya no tenemos la función *suma* que restringía el tipo de las dos variables a *int*. En el caso de la instancia *a*, sus variables *@x* y *@y* son de tipo *int*.

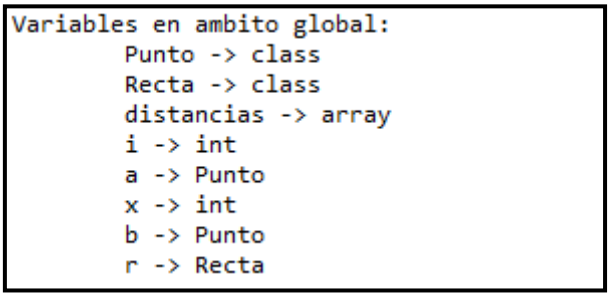


FIGURA 42: RESULTADOS DEL ANALISIS EN DE LA PRUEBA 2

Podemos ver también como quedan también los tipos en general, sin **ningún error**. Consideramos interesante marcar *Punto* y *Recta* como clases en el ámbito general para aportar la información de forma simple y rápida al usuario.

4.3 PRUEBA 3

Para la tercera prueba decidimos usar un código de un nivel más avanzado, y puesto que el analizador está escrito en Ruby, pensamos que sería una buena idea ejecutar el análisis sobre el propio código del analizador. En este caso tenemos dos ficheros distintos, uno con las estructuras creadas para almacenar la información de las cuales ya hemos hablado en el apartado de desarrollo y otro con el analizador principal que hace uso de todas esas clases. Para esta prueba hemos decidido ejecutar el analizador sobre el fichero con todas las estas clases.

Vamos a ver primero varios ejemplos de cómo ha analizado las **clases** y después probaremos a instanciar algunas de estas clases para comprobar si se realiza correctamente la propagación.

```
16  class ArrayDef
17    def initialize (nombre)
18      @nombre = nombre
19      @tipos = Array.new
20    end
21
22    def anadeTipo (tipo)
23      if !@tipos.include?(tipo)
24        @tipos << tipo
25      end
26    end
27
28    def setTipos (tipos)
29      @tipos = tipos
30    end
31
32    def getTipos
33      return @tipos
34    end
35
36    def getNombre
37      return @nombre
38    end
39
40    def daInfo
41      puts "Array de nombre #{@nc
42    end
43  end
44
45
46  class VariableDef
47
48    def initialize (nombre)
49      @nombre = nombre
50      @tipos = Array.new
```

```
Nombre de la clase: ArrayDef
Variables:
        Nombre: @nombre
        Tipo(s): [nil]
        Nombre: @tipos
        Tipo(s): [:array]
Arrays:
Funciones:
        Nombre de la funcion: initialize
        Argumentos:
                Nombre: nombre
                Tipo(s): [nil]
                Conexiones: [:@nombre]

        Nombre de la funcion: anadeTipo
        Argumentos:
                Nombre: tipo
                Tipo(s): [nil]

        Nombre de la funcion: setTipos
        Argumentos:
                Nombre: tipos
                Tipo(s): [nil]
                Conexiones: [:@tipos]

        Nombre de la funcion: getTipos
        Argumentos:
        Return de tipo: [:array]

        Nombre de la funcion: getNombre
        Argumentos:
        Return de tipo: [nil]

        Nombre de la funcion: daInfo
        Argumentos:
```

FIGURA 43: EJEMPLO DE FUNCIONAMIENTO DEL ANALIZADOR EN LA PRUEBA 3

En la **Figura 42** se puede observar un ejemplo muy claro de cómo se ha hecho el análisis de la clase ***ArrayDef*** sin ningún tipo de fallo. Vemos cómo es capaz de reconocer que *@tipos* es un *array* y hacer correctamente las conexiones entre los argumentos y las variables de instancia al igual que con los *returns*.

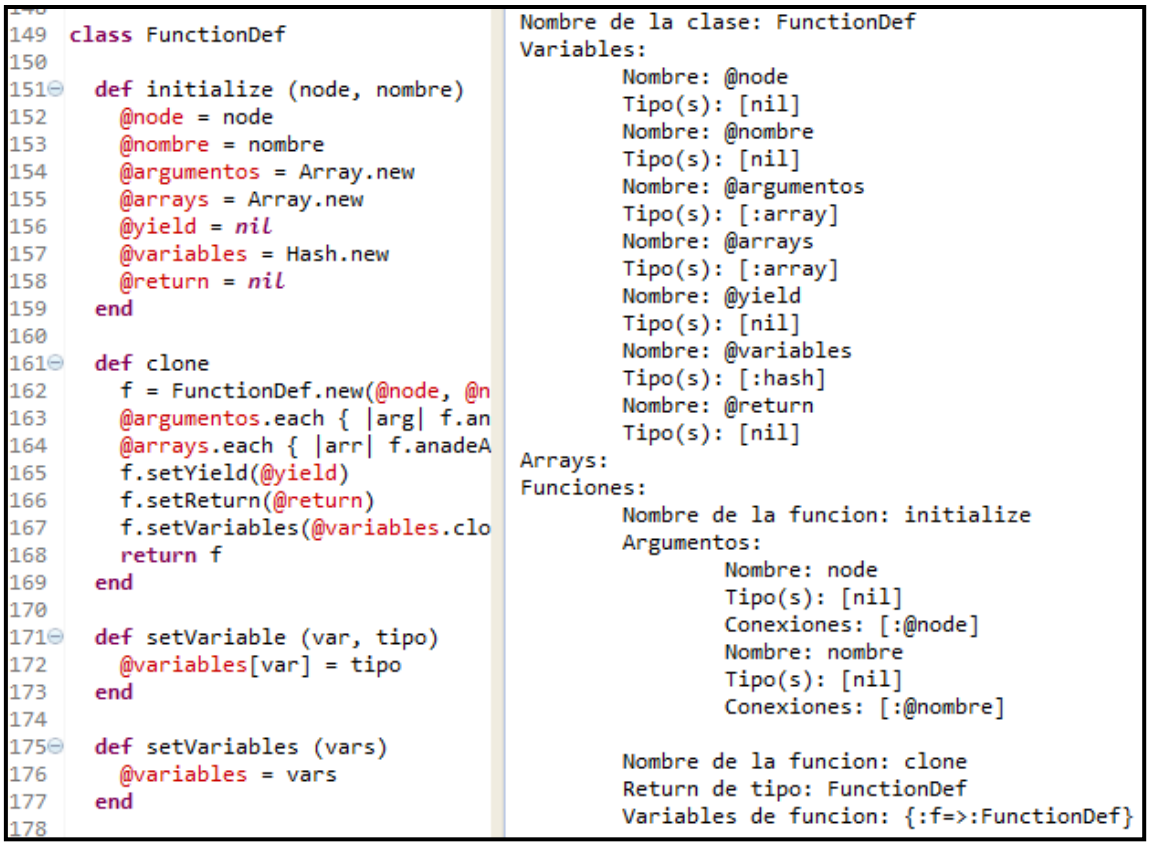
Lo único remarcable son aquellos tipos que quedan marcados como *nil* ya que hasta que no se cree una instanciación de la clase el analizador no será capaz de asignarles ningún tipo.

```
16  class ArrayDef
17    def initialize (nombre)
18      @nombre = nombre
19      @tipos = Array.new
20    end
21
22    def anadeTipo (tipo)
23      if !@tipos.include?(tipo)
24        @tipos << tipo
25      end
26    end
27
28    def setTipos (tipos)
29      @tipos = tipos
30    end
31
32    def getTipos
33      return @tipos
34    end
35
36    def getNombre
37      return @nombre
38    end
39
40    def daInfo
41      puts "Array de nombre #{@nomb
42    end
43  end
44
45  probando = ArrayDef.new("Prueba")
46  probando.anadeTipo(:int)
47
48
49  class VariableDef
```

```
Clases Instanciadas:
Nombre: probando
Clase de tipo ArrayDef
Variables:
        Nombre: @nombre
        Tipo(s): [:str]
        Nombre: @tipos
        Tipo(s): [:array]
Arrays:
Funciones:
        Nombre de la funcion: initialize
        Argumentos:
                Nombre: nombre
                Tipo(s): [:str]
                Conexiones: [:@nombre]

        Nombre de la funcion: anadeTipo
        Argumentos:
                Nombre: tipo
                Tipo(s): [:sym]

        Nombre de la funcion: setTipos
        Argumentos:
                Nombre: tipos
                Tipo(s): [nil]
                Conexiones: [:@tipos]

        Nombre de la funcion: getTipos
        Argumentos:
        Return de tipo: [:array]

        Nombre de la funcion: getNombre
        Argumentos:
        Return de tipo: [:str]
```

FIGURA 44: OTRO EJEMPLO DE FUNCIONAMIENTO DEL ANALIZADOR EN LA PRUEBA 3

En la **Figura 43** podemos ver la misma clase que en la figura anterior pero esta vez hemos hecho una instancia de ella llamada probando cuyo análisis se puede ver en la derecha de la figura. La diferencia está en que ahora el argumento de *initialize* es una variable de tipo *str*, y dicho tipo se ha propagado tanto a *@nombre* como al *return* de la función *getNombre*. También podemos ver que el argumento tipo de la función *anadeTipo* es de tipo *sym*.



```
149 class FunctionDef
150
151   def initialize (node, nombre)
152     @node = node
153     @nombre = nombre
154     @argumentos = Array.new
155     @arrays = Array.new
156     @yield = nil
157     @variables = Hash.new
158     @return = nil
159   end
160
161   def clone
162     f = FunctionDef.new(@node, @n
163     @argumentos.each { |arg| f.an
164     @arrays.each { |arr| f.anadeA
165     f.setYield(@yield)
166     f.setReturn(@return)
167     f.setVariables(@variables.clo
168     return f
169   end
170
171   def setVariable (var, tipo)
172     @variables[var] = tipo
173   end
174
175   def setVariables (vars)
176     @variables = vars
177   end
178
```

```
Nombre de la clase: FunctionDef
Variables:
        Nombre: @node
        Tipo(s): [nil]
        Nombre: @nombre
        Tipo(s): [nil]
        Nombre: @argumentos
        Tipo(s): [:array]
        Nombre: @arrays
        Tipo(s): [:array]
        Nombre: @yield
        Tipo(s): [nil]
        Nombre: @variables
        Tipo(s): [:hash]
        Nombre: @return
        Tipo(s): [nil]
Arrays:
Funciones:
        Nombre de la funcion: initialize
        Argumentos:
                Nombre: node
                Tipo(s): [nil]
                Conexiones: [:@node]
                Nombre: nombre
                Tipo(s): [nil]
                Conexiones: [:@nombre]

        Nombre de la funcion: clone
        Return de tipo: FunctionDef
        Variables de funcion: {:f=>:FunctionDef}
```

FIGURA 45: EJEMPLO DISTINTO DE RESULTADO DEL ANALIZADOR EN LA PRUEBA 3

En la **Figura 44** tenemos un ejemplo de una clase más compleja, en este caso la utilizada para almacenar las funciones. Podemos ver cómo el analizador es capaz de asignar todos los tipos posibles a las variables de instancia, así como las conexiones de la función *initialize*. También se puede observar cómo ha procesado la función clone asignando los tipos correctos a la variable *f* y su valor de retorno.

5 CONCLUSIONES Y TRABAJO FUTURO

5.1 RESULTADOS Y LIMITACIONES

Como hemos podido ver a lo largo de todo este trabajo, nuestro analizador sintáctico de código Ruby es capaz de **inferir el tipo** de la gran mayoría de las variables de programa, así como dar la **información** de todos los objetos creados.

También hemos visto que nuestro analizador tiene ciertas **limitaciones** que hacen que no sea capaz de resolver los tipos de todas las variables. En concreto hay cuatro limitaciones que ya hemos explicado en apartados anteriores:

La primera limitación es a la hora de **propagar ciertos tipos** entre funciones. Hay ciertos casos en los que no es seguro asignar un tipo fijo a ciertas variables de función solo por una llamada a ese método, ya que no hay nada que no garantice que más adelante se vuelva a llamar a ese método con otros argumentos de tipo distinto. Aun así esto ocurre en casos muy concretos en los que esas funciones no son llamadas o las clases no son instanciadas en el código y por lo tanto no se le puede asignar un tipo siguiendo la filosofía de *Duck Typing* (por ejemplo si a esas variables se les aplica una suma podríamos inferir que son de tipo `int`).

La segunda limitación aparece cuando en el código de programa se hace uso de **funciones propias del lenguaje** Ruby las cuales siempre devuelven tipos concretos. Al haber una cantidad incontrolable de este tipo de funciones, nuestro analizador no es capaz de abarcarlas todas por completo con lo cual variables cuyos tipos dependan únicamente de este tipo de funciones se quedarán sin tipar. Sin embargo, sería técnicamente posible solventar este problema como veremos más adelante en el apartado de trabajo futuro.

La tercera limitación tiene que ver con algo propio de los lenguajes dinámicos y es la **metaprogramación**. En concreto Ruby tiene un método llamado *eval* el cual ejecuta un bloque de código que se le haya pasado como parámetro. Al ser ese código una cadena de texto

es imposible analizarla e inferir los tipos de las variables que pudiera tener incluidas.

Finalmente, la cuarta limitación es la más simple de todas y es que como ya comentamos en el apartado de pruebas, para que el analizador funcione correctamente el código debe estar organizado de tal manera que no haya ninguna instanciación de una clase que no haya sido definida previamente. Nuestro analizador procesa el texto **secuencialmente** por lo tanto no es capaz de reconocer como clases ciertas instancias si no las ha procesado hasta ese momento.

En general nuestro analizador es una herramienta bastante útil que sirve como **asistencia** para código programado por desarrolladores que están en proceso de aprendizaje del lenguaje (cada vez más común gracias a la creciente popularidad de Ruby) como para código más avanzado propio de programadores más expertos que quieren tener una herramienta que les indique cómo han construido el programa o el tipo de cierta variable, mejor que buscarla a lo largo de todo el código.

5.2 TRABAJO FUTURO

Para analizar las posibles mejoras a nuestra API, tenemos que fijarnos en las limitaciones que comentamos en el apartado anterior.

Sobre la primera limitación en cuanto a la propagación de tipos y variables sin tipo concreto, una solución posible sería fijarnos en la que proponía Ecstatic, crear un **simulador** de código Ruby para ver cómo se comportaría el programa al ejecutarlo y gracias a eso establecer todos los tipos que no se podrían conocer de otra forma. Esta solución presenta dos inconvenientes, el primero es que no se podría considerar completamente un análisis estático ya que el simulador significa cierto tipo de ejecución, y el segundo es que precisamente esa simulación acarrea una disminución en cuanto a rendimiento ya que el análisis será significativamente más lento.

En cuanto a la segunda limitación, se trata simplemente de una cuestión práctica. Analizar e incluir todas las funciones de las librerías propias de Ruby a nuestro analizador llevaría una gran cantidad de tiempo. La otra solución sería una aplicación intermedia entre las librerías y el analizador para el **intercambio de información**. No sería realmente complicado pero sí que sería costoso y además también veríamos mermado el rendimiento de nuestra API al tener

que analizar la gran cantidad de funciones que estarían incluidas siempre que encontráramos una llamada a un método en el código a analizar salvo que las fuéramos almacenando en una especie de caché según aparecen.

La tercera limitación relativa a la instrucción *eval* es la más complicada de solventar ya que al fin y al cabo se trata de analizar una parte de código escrita en forma de cadena de texto dentro del propio código ya pasado por el parser. Para ello tendríamos que caer en una especie de **recursión** ya que tendríamos que volver al análisis sintáctico una vez dentro del análisis estático de tipos y aun así tendríamos más problemas ya que ese código insertado depende del resto del programa y viceversa, siendo posible su análisis completo únicamente en tiempo de ejecución.

Podemos entonces concluir que aunque nuestro analizador no sea capaz de inferir completamente el tipo de todas las variables, es una herramienta ligera capaz de realizar un **tipado estático** en muy poco tiempo. En un futuro se podrían llegar a incluir todas las funciones de las librerías estándar de Ruby para completarla aun más aunque habría que valorar el beneficio que supondría frente a la utilidad real y sobre todo a la caída de rendimiento que supondría.

En cuanto a la aplicación en sí, se trata de una primera versión que podría hacer las veces de prototipo pero podría marcar el camino hacia un analizador más completo gracias a las mejoras comentadas y que podría acabar **integrándose en entornos de desarrollo** como Eclipse o Netbeans para estar a la disposición de todos los usuarios.

BIBLIOGRAFÍA

[1] "About Ruby." [Online]. Available: https://www.Ruby-lang.org/en/about/ [Accesed: 29-Mar.2015]

[2] "Differences between compiled and interpreted languajes." [Online]. Available: http://www.codeproject.com/Articles/696764/Differences-between-compiled-and-Interpreted-Langu [Accesed: 2-Apr-2015]

[3] Laurence Tratt. *Dynamically typed languages*, Advances in Computers, vol. 77, pages 149-184, July 2009

[4] "Duck Typing: Ruby study notes". [Online]. Available: http://rubylearning.com/satishtalim/duck_typing.html [Accesed: 2-Apr-2015]

[5] B. M. Ren, J. Toman, T. S. Strickland, and J. S. Foster. *The Ruby type checker*. In Proceedings of the 28th Symposium on Applied Computing (SAC), 2013.

[6] J. An, A. Chaudhuri, and J. S. Foster. *Static type inference for Ruby*. In Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering, 2009.

[7] M. Madsen, P. Sørensen, and K. Kristensen. *Ecstatic – type inference for Ruby using the cartesian product algorithm*. Master's thesis, Aalborg University, 2007.

[8] "wimrijnders/jParser". [Online]. Available: https://github.com/wimrijnders/jParser [Accesed: 7-Apr-2015]

[9] "whitequark/parser". [Online]. Available: https://github.com/whitequark/parser [Accesed: 7-Apr-2015]

GLOSARIO

API *Application Programming Interface*. Capa de abstracción en la que una biblioteca recoge toda su funcionalidad para ser utilizada por otro software.

AST *Abstract Syntax Tree*. Árbol de sintaxis abstracta el cual contiene la información de un código Ruby en cada uno de sus nodos.

CPA *Cartesian Product Algorithm*. Algoritmo de producto cartesiano utilizado para inferir tipos durante el análisis.

Lenguaje dinámico Aquel lenguaje de programación que acepta modificaciones en tiempo de ejecución.

Lenguaje estático Aquel lenguaje de programación que no acepta modificaciones a partir de su compilación

Nodo Cada una de las partes en las que se divide el AST y que contiene información del código Ruby.

Objeto de programa Cada uno de los elementos básicos del código como clases, funciones, variables, etc.

Parser Herramienta utilizada para transformar un código Ruby en un AST mediante un análisis sintáctico.

Tipado Asignación de un tipo a cierta variable de programa.